Softwaretechnik

Prof. Dr. Rainer Koschke

Fachbereich Mathematik und Informatik
Arbeitsgruppe *Softwaretechnik*
Universität Bremen

Sommersemester 2006

Überblick I

Überblick II

8. Software-Produktlinien

9. Empirische Softwaretechnik

Vorbemerkungen

Übersicht

- Metriken
- Kosten- und Aufwandsschätzung
- Entwicklungsprozesse
- SESAM-Schulung
- Komponentenbasierte Entwicklung
- Entwurfsmuster
- Software-Architektur
- Software-Produktlinien

SESAM



Übungen und Ressourcen

Dozent:

- http://www.informatik.uni-bremen.de/~koschke/
- Sprechstunde nach Vereinbarung

Ressourcen:

- annotierte Folien unter http://www.informatik.uni-bremen.de/st/lehredetails.php?id=&lehre_id=406
- Videoaufzeichnungen unter http://mlecture.uni-bremen.de/
- News unter Stud.IP unter http://elearning.uni-bremen.de

Übungen:

- Übungen ca. alle zwei Wochen alternierend zur Vorlesung
- Übungsblatt im Netz

Meine Grundsätze der Leistungsbewertung

- Übungen sollten keine Prüfungsleistungen sein
- praktisch anwenden ist besser als wiederkäuen
- umfassendes Lernen ist besser als punktuelles
- Noten müssen individuellen Beitrag wiedergeben
- die Form der Prüfungsleistung muss einheitlich sein
- es muss einen Unterschied zwischen Modulprüfung und Schein geben

Formen der Prüfungsleistungen

(a) mündliche Prüfung,
(b) Klausurarbeit,
(c) Bearbeitung von Übungsaufgaben mit Fachgespräch,
(d) Bearbeitung von Praktikums- bzw. Laboraufgaben mit Fachgespräch,
(e) mündlicher Vortrag mit schriftlicher Ausarbeitung (Referat), optional mit Fachgespräch,
(f) umfangreiche schriftliche Ausarbeitung (Hausarbeit) mit Fachgespräch,
(g) Abschlussarbeit.

Scheinbedingungen

Anerkennung durch mündliche Prüfung:

- 30 minütige mündliche Prüfung über den Stoff der Vorlesung
- Übungsaufgaben bearbeiten lohnt sich
- Wiederholungsfragen beantworten lohnt sich

Ansonsten:

- Fachgespräch (zählt zu 30%)
- Übungs-/Praktikumsaufgaben (zählen zu 70%):
  - Kosten- und Aufwandsschätzung für System *S*
  - Vorschlag eines Prozessmodells für die Entwicklung von *S*
  - Architekturentwurf bzw. -analyse für *S*

  wobei *S* eine Online-Bibliographie für wissenschaftliche Referenzen ist, die im Software-Projekt gerade durch eine Neuentwicklung ersetzt wird:
  `http://www.iste.uni-stuttgart.de/ps/reengineering/`

Online-Bibliographie: Startseite

Reengineering Bibliography - Mozilla

File Edit View Go Bookmarks Tools Window Help

http://www.iste.uni-stuttgart.de/ps/reengineering/index.html Search

Home Bookmarks The Mozilla Org... Latest Builds

IEEE COMPUTER SOCIETY
Technical Council on Software Engineering

Welcome to the Reengineering Bibliography

This annotated bibliography provides information on software reengineering.

This reengineering bibliography is an initiative of several people. In their rare spare time they collected bibliographic entries and merged them into this elaborate bibliography. Probably you have done the same, then please mail us so that we can add your references all at once. Maybe you have contributed to the field of reengineering yourself. Since the originators lack the time to continuously update this bibliography we ask you to add your own references. This is a very simple process when using our forms.

Supporters

This annotated bibliography is supported by Bauhaus Software Technologies, a company providing services and tools to support reengineering, maintenance, and evolution of existing system. Please visit the home page of our supporters.

BAUHAUS
SOFTWARE
TECHNOLOGIES

Sites

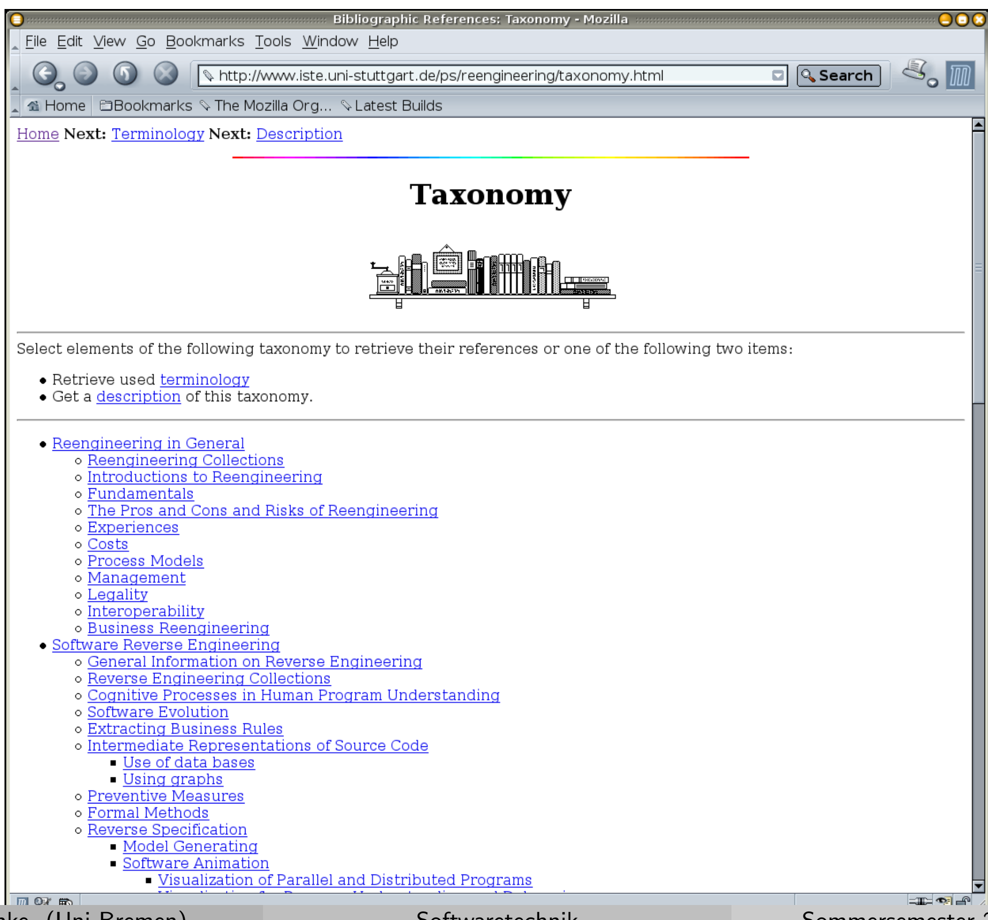
The reengineering bibliography is available at two sites. Choose the one closest to you:

- University of Stuttgart, Germany.
- Georgia Institute of Technology, Atlanta, GA, USA.

Contents

- Introduction
- Leaflet (Postscript) Print and fold it and it will be a valuable guide in searching and adding references.
- News
- Adding your references
- Searching for references..
  - ..via a taxonomy of reengineering-related terms.
  - ..via a simple search form that allows retrievals by specifying author, title, keywords, abstract, year, etc.
- The bibliography as a BibTeX database via FTP
- Taxonomy

Online-Bibliographie: Taxonomiesuche



Online-Bibliographie: Suche nach Eigenschaften

Bibliography Search - Mozilla

File Edit View Go Bookmarks Tools Window Help

http://www.iste.uni-stuttgart.de/ps/reengineering/simple+search.html Search

Home Bookmarks The Mozilla Org... Latest Builds

Home **Next:** Regular Expressions

Search for References

Fill out this form (you can leave out fields), press the button below, and you will receive all the references that contain the regular expressions you filled in. Note that the search is case-insensitive and that the entries to retrieve must have **all** specified attributes.

author

title

keywords

abstract

Start searching | Clear all inputs

koschke@informatik.uni-stuttgart.de (Feedback).
Copyright © 1997 University of Stuttgart, Germany. $Revision: 1.6 $
Last modified: Mon Jun 22 16:26:16 MET DST 1998

Done

Literaturreferenzen I

Allgemeine Literatur zur Softwaretechnik



- Sommerville (2004)
- Pressman (1997)

Software-Metriken

- Fenton und Pfleeger (1998)

Aufwand- und Kostenschätzung

- Boehm u. a. (2000)

Software-Entwicklungsprozesse

- Beck (2000)
- Kruchten (1998)
- auch: Sommerville (2004); Pressman (1997)

Literaturreferenzen II

Komponentenbasierte Entwicklung

- Szyperski u. a. (2002)

Software-Architektur

- Bass u. a. (2003)
- Hofmeister u. a. (2000)
- Buschmann u. a. (1996)

Entwurfsmuster

- Gamma u. a. (2003)

Software-Produktlinien

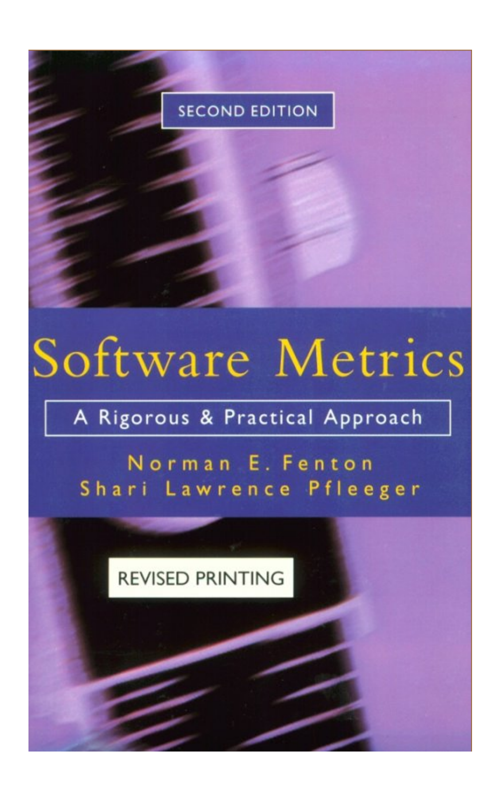
- Clements und Northrop (2001)

Software-Metriken

Lernziele

- Verstehen, was eine Software-Metrik ist
- die Einsatzmöglichkeiten von Metriken kennen
- Metriken beurteilen und auswählen können

Literatur



– Fenton und Pfleeger (1998)

Bedeutung des Messens

“To measure is to know.” Clerk Maxwell, 1831-1879

“A science is as mature as its measurement tools.”
Louis Pasteur, 1822-1895

„Miss alles, was sich messen lässt, und mach alles messbar, was sich nicht messen lässt.“ Galileo Galilei, 1564-1642

„Messen können Sie vieles, aber das Angemessene erkennen können Sie nicht.“ Hans Gadamer

Bedeutung des Messens

“To measure is to know.” Clerk Maxwell, 1831-1879
“A science is as mature as its measurement tools.” Louis Pasteur, 1822-1895
„Miss alles, was sich messen lässt, und mach alles messbar, was sich nicht messen lässt.“ Galileo Galilei, 1564-1642
„Messen können Sie vieles, aber das Angemessene erkennen können Sie nicht.“ Hans Gadamer

Messen spielt in allen Ingenieurswissenschaften eine wichtige Rolle.
Galilei: Ziel der Wissenschaft; durch Messung zu verständlicheren und nachprüfbaren Konzepten/Ergebnissen kommen.

Definitionen: Maß, Messen, Metrik

Definition

Maß:
Abbildung von einem beobachteten (empirischen) Beziehungssystem auf ein numerisches Beziehungssystem

Abbildung von Eigenschaften von Objekten der realen Welt auf Zahlen oder Symbole

Definition

Messen: Anwendung eines Maßes auf ein Objekt

Definition

Metrik: Abstandsmaß (math.)

Definitionen für Software-Metriken

"A quantitative measure of the degree to which a system, component, or process possesses a given variable."
– IEEE Standard Glossary

"A software metric is any type of measurement which relates to a software system, process or related documentation."
– Ian Summerville

Messen und Softwaretechnik

- Beschreibung: kompakt und objektiv
- Beurteilung: Vergleich, Verbesserungen
- Vorhersage: geordnete Planung, Verbesserungen

Fragen bei Maßen

Worüber wir uns bei der Definition von Metriken Gedanken machen müssen:

- Repräsentanz — Darstellung als Zahl sinnvoll möglich?
- Eindeutigkeit — viele Abbildungen möglich
- Bedeutung — erhalten bei Transformationen
- Skalierung — welche Skala?

Fragen bei Maßen

Worüber wir uns bei der Definition von Metriken Gedanken machen müssen:

- Repräsentanz — Darstellung als Zahl sinnvoll möglich?
- Eindeutigkeit — viele Abbildungen möglich
- Bedeutung — erhalten bei Transformationen
- Skalierung — welche Skala?

There are three important questions concerning representations and scales:

1. How do we determine when one numerical relation system is preferable to another?
2. How do we know if a particular empirical relation system has a representation in a given numerical relation system?
3. What do we do when we have several different possible representations (and hence many scales) in the same numerical relation system?

Question 2 is known as the representation problem.

Fenton und Pfleeger (1998)

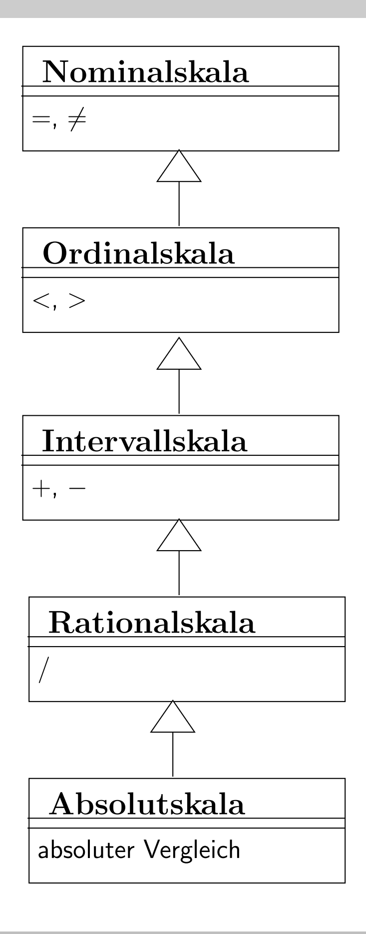
Skalen

1. „20 Prozent Verbesserung der Qualität"
2. „Dieser Kunde ist doppelt so zufrieden wie jener"
3. „Heute doppelt so warm wie gestern"
   (Temperatur gestern: 10°C; heute: 20°C)

1. Was ist Qualität Null?
2. Wie zufrieden sind Sie denn?
3. $10°C \rightarrow 20°C \mathrel{\hat{=}} +3{,}5\%$
   denn $10°C = 283$ Kelvin, $20°C = 293$ Kelvin

$\rightarrow$ Skala?

Skalenhierarchie



Skalenhierarchie – Nominalskala

1. Nominalskala
   - ungeordnete 1:1 Abbildung
   - Transformationen: beliebige 1:1
   - Operationen: $=$, $\neq$
   - Statistiken: Häufigkeit
   - Beispiel: Programmiersprachen

Ada C C++ Java ...

Skalenhierarchie – Ordinalskala

2. Ordinalskala
   - dazu: vollständige Ordnung
   - Transformationen: streng monoton steigend
   - Operationen: $<$, $>$
   - Statistiken: Median
   - Beispiel: Prioritäten

$$\text{niedrig} < \text{mittel} < \text{hoch}$$

Skalenhierarchie – Intervallskala

3. Intervallskala
   - dazu: Distanzfunktion
   - Transformationen: $M' = aM + b \quad (a > 0)$
   - Operationen: $+$, $-$
   - Statistiken: Mittelwert, Standardabweichung
   - Beispiel: Temperatur

$$T_{Celsius} = \frac{5}{9} \cdot (T_{Fahrenheit} - 32)$$

Definition Metrik

Metrik: Distanzfunktion $d : A \times A \rightarrow IR$, mit:

- $d(a, b) \geq 0 \; \forall a, b \in A, \quad d(a, b) = 0 \Leftrightarrow a = b$
- $d(a, b) = d(b, a) \; \forall a, b \in A$
- $d(a, c) \leq d(a, b) + d(b, c) \; \forall a, b, c \in A$

Skalenhierarchie – Rationalskala

4. Rationalskala

- dazu: Maßeinheit, Nullpunkt
- Transformationen: $M' = aM \quad (a > 0)$
- Operationen: /
- Statistiken: geom. Mittel, Korrelation
- Beispiel: Länge

$$L_{Meter} = L_{Meilen} \cdot 1609$$

Skalenhierarchie – Rationalskala

4. Rationalskala
- dazu: Maßeinheit, Nullpunkt
- Transformationen: $M' = aM \quad (a > 0)$
- Operationen: /
- Statistiken: geom. Mittel, Korrelation
- Beispiel: Länge

$$L_{Meter} = L_{Meilen} \cdot 1609$$

Das geometrische Mittel zwischen zwei Zahlenwerten ist: $\sqrt{f1} \cdot f2$
Das arithmetische Mittel zwischen zwei Zahlwerten ist: $(f1 + f2)/2$

Skalenhierarchie – Absolutskala

5. Absolutskala

- Metrik steht für sich selbst, kann nicht anders ausgedrückt werden
- Transformationen: nur die Identität $M' = M$
- Operationen: absoluter Vergleich; d.h
  - es existiert ein natürlicher Nullpunkt
  - und Maßeinheit ist natürlich gegeben (d.h. im weitesten Sinne 'Stück')
- Beispiele:
  - Zähler: Anzahl Personen in einem Projekt
  - Wahrscheinlichkeit eines Fehlers
  - LOC für Anzahl Codezeilen
  - **nicht:** LOC für Programmlänge

Gütekriterien für Metriken

- Objektivität: unabhängig vom Messenden
- Validität: misst, was sie vorgibt zu messen
- Zuverlässigkeit: Wiederholung liefert gleiche Ergebnisse
- Nützlichkeit: hat praktische Bedeutung
- Normiertheit: es gibt eine Skala für die Messergebnisse
- Vergleichbarkeit: mit anderen Maßen vergleichbar
- Ökonomie: mit vertretbaren Kosten messbar

– Balzert (1997)

Gütekriterien für Metriken
- Objektivität: unabhängig vom Messenden
- Validität: misst, was sie vorgibt zu messen
- Zuverlässigkeit: Wiederholung liefert gleiche Ergebnisse
- Nützlichkeit: hat praktische Bedeutung
- Normiertheit: es gibt eine Skala für die Messergebnisse
- Vergleichbarkeit: mit anderen Maßen vergleichbar
- Ökonomie: mit vertretbaren Kosten messbar
– Balzert (1997)

(Güte entspr. Qualität)
Objekt.: kein subjektiver Einfluss durch Messenden möglich
Valid.: misst wirklich das, was sie vorgibt zu messen
Zuverl.: Wiederholung liefert gleiche Ergebnisse
Nützl.: hat praktische Bedeutung
Norm.: es gibt eine Skala für die Messergebnisse
Vergl.: mit anderen Maßen vergleichbar
Ökon.: mit vertretbaren Kosten messbar

Vorgehensweise

1. Definition eines Maßes
   - Zielbestimmung
   - Modellbildung
   - Skalentypbestimmung
   - Maßdefinition
2. Validierung des Maßes
   - Interne Validierung
   - Externe Validierung
3. Anwendung des Maßes
   - Konkretes Modell bilden
   - Messung
   - Interpretation
   - Schlussfolgerung

Validierung von Maßen

Interne Validierung:

Nachweis, dass ein Maß eine gültige numerische Charakterisierung des entsprechenden Attributs ist, durch

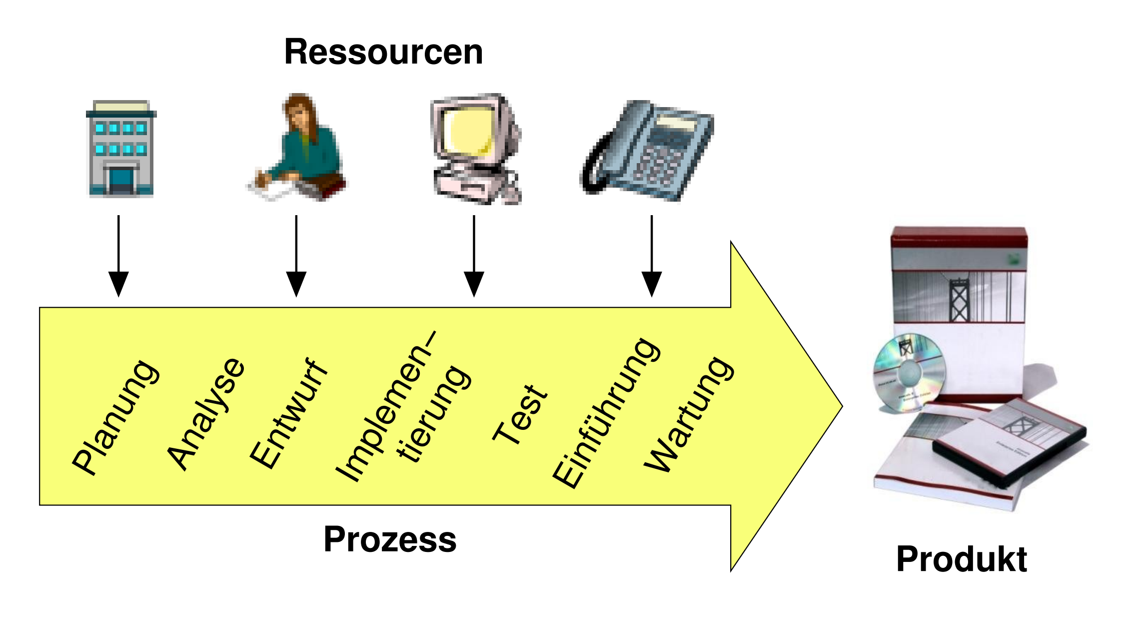
- Nachweis der Erfüllung der Repräsentanzbedingung
- und Prüfung des Skalentyps

Externe Validierung → Vorhersagemodell:

- Hypothese über Zusammenhang zwischen zwei Maßen
- Erfassung der Meßwerte beider Maße auf gleicher Testmenge
- Statistische Analyse der Ergebnisse
  → Bestimmung von Parametern
  → Prüfung der Allgemeingültigkeit

Klassifikation von Softwaremetriken

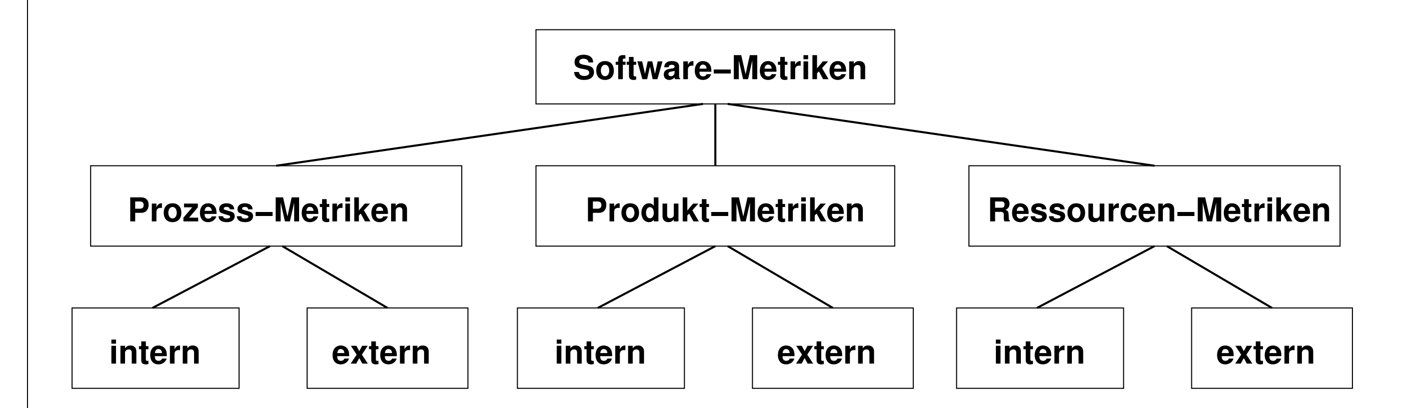
- Was: Ressource/Prozess/Produkt
- Wo: intern/extern (isoliert/mit Umgebung)
- Wann: in welcher Phase des Prozesses
- Wie: objektiv/subjektiv, direkt/abgeleitet



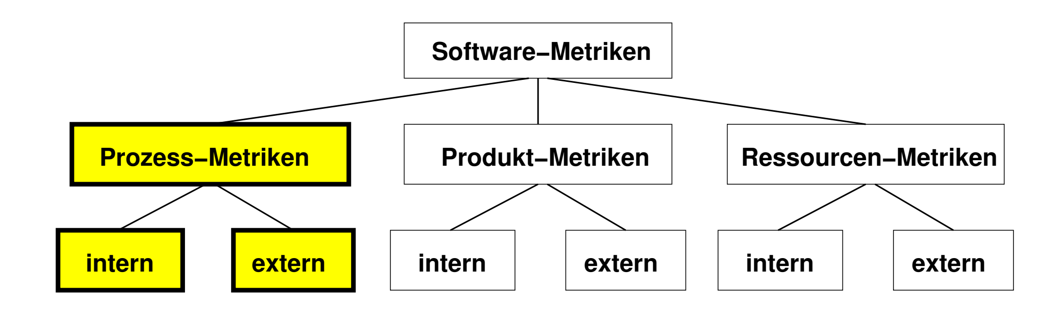
Klassifikation von Softwaremetriken
Was: Ressource/Prozess/Produkt
Wo: intern/extern (isoliert/mit Umgebung)
Wann: in welcher Phase des Prozesses
Wie: objektiv/subjektiv, direkt/abgeleitet

Bei den Metriken unterscheidet man zwischen *internen* und *externen* Metriken. Eine interne Metrik ist darüber definiert, dass sie nur Eigenschaften innerhalb des untersuchten Objektes misst, wohingegen externe Metriken die Interaktion des Objektes mit seiner Umgebung berücksichtigen.

Klassifikation nach Fenton und Pfleeger (1998)



Prozessmetriken



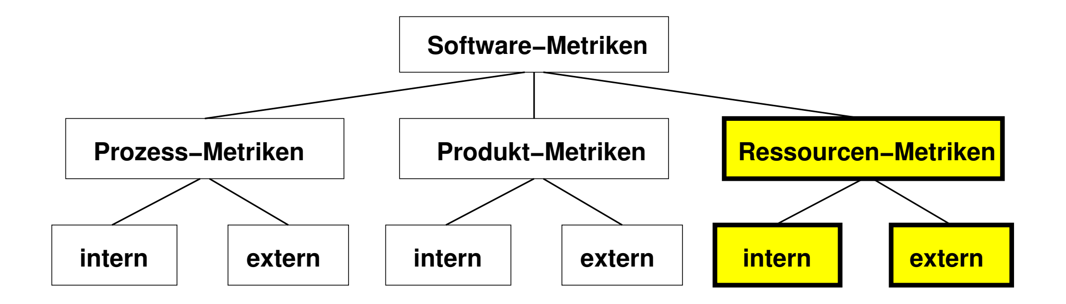
intern:

- Zeit/Dauer
- Aufwand
- Anzahl von Ereignissen
  z.B. Fehler, Änderungen

extern:

- Qualität
- Kontrollierbarkeit
- Stabilität
- Kosten

Ressourcenmetriken



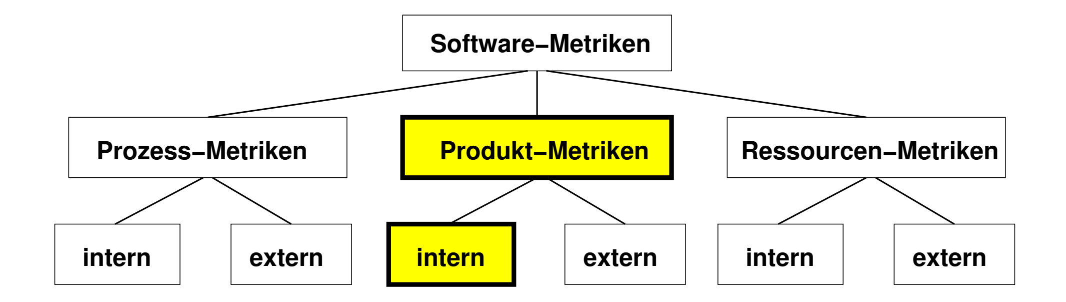
intern:

- Personal (Alter, Lohn)
- Teamgröße/-struktur
- Produktionsmaterialien
- Werkzeuge, Methoden

extern:

- Produktivität
- Erfahrung
- Kommunikation
- ...

Produktmetriken – intern



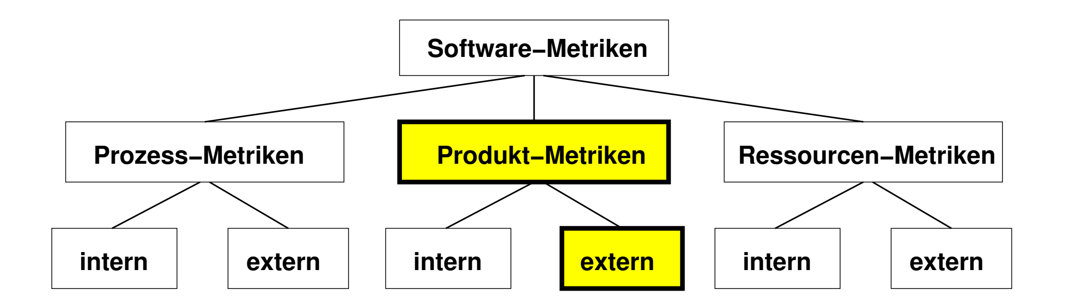
Größe:

- LOC
- Halstead
- Function Points
- Bang (DeMarco)

Komplexität:

- McCabe Cyclomatic Complexity
- Kontrollflussgraph
- Datenfluss
- OO-Metriken

Produktmetriken – extern



- Zuverlässigkeit
- Verständlichkeit
- Benutzerfreundlichkeit
- Performanz
- Portierbarkeit
- Wartbarkeit
- Testbarkeit
- . . .

Produktmetriken – intern

Vorteil: automatische Erfassung

Die Klassiker:

- LOC - Lines Of Code
- Halstead (1977)
- McCabe (1976)
- OO-Metriken (Chidamber und Kemerer 1994)

Größenmetriken – LOC

Lines of code (LOC)

+ relativ einfach messbar
+ starke Korrelation mit anderen Maßen
– ignoriert Komplexität von Anweisungen und Strukturen
– schlecht vergleichbar
- abgeleitet: Kommentaranteil

Physical source lines (COCOMO 2.0)

When a line or statement contains more than one type, classify it as the type with the highest precedence.

Statement type	Precedence	Included
Executable	1	$\surd$
Nonexecutable		
Declarations	2	$\surd$
Compiler directives	3	
Comments		
On their own lines	4	
On lines with source code	5	
Banners and non-blank spacers	6	
Blank (empty) comments	7	
Blank lines	8	

Physical source lines (COCOMO 2.0)

How produced	Included
Programmed	$\surd$
Generated with source code generators	
Converted with automated translators	$\surd$
Copied or reused without change	$\surd$
Modified	$\surd$
Removed	

Physical source lines (COCOMO 2.0)

Origin	Include
New work: no prior existence	$\surd$
Prior work: taken or adapted from	$\surd$
A previous version, build, or release	$\surd$
Commercial off-the-shelf software (COTS), other than libraries	
Government furnished software (GFS), other than reuse libraries	
Another product	
A vendor-supplied language support library (unmodified)	
A vendor-supplied operating system or utility (unmodified)	
A local or modified language support library or operating system	
Other commercial library	
A reuse library (software designed for reuse)	$\surd$
Other software component or library	$\surd$

Anwendungen

- Beurteilung des aktuellen Zustands
  - Projektüberwachung
  - Produktivität
  - Softwarequalität
  - Prozessqualität (CMM)
- Vorhersage des zukünftigen Zustands
  - Aufwandsabschätzung
  - Prognose für Wartungskosten

Probleme

- Datenerfassung sehr aufwendig, zunächst wenig Nutzen
- Datenerfassung nicht konsistent
- Teilweise Messungen schwierig durchführbar
- Zweck der Messungen muss klar sein
- Integration der Datenerfassung in den normalen Arbeitsprozess
- Metriken müssen wohldefiniert und validiert sein
- Beziehungen zwischen Metriken müssen definiert sein
- Gefahr der Fehlinterpretation

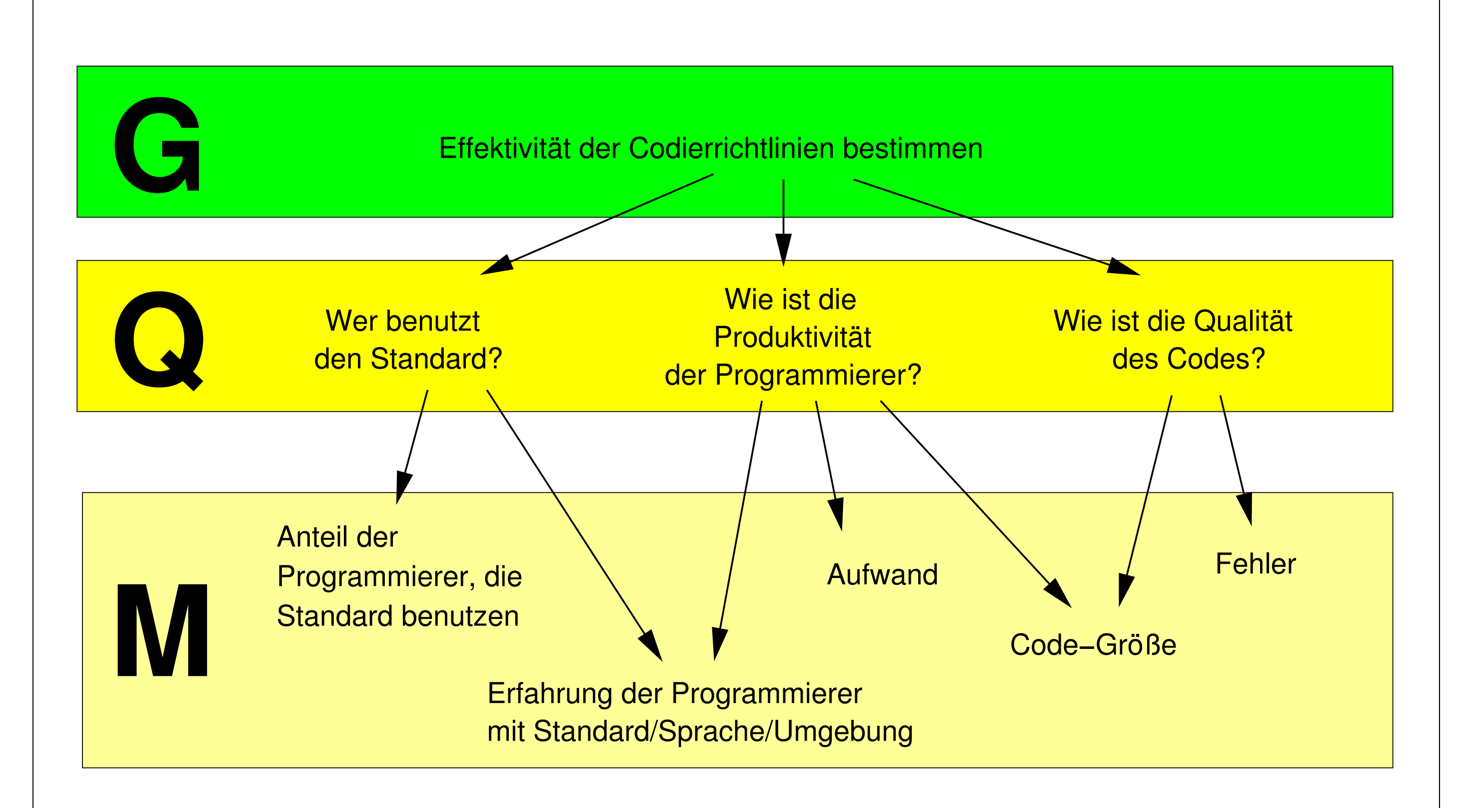
Zielorientiertes Messen

GQM (Goal-Question-Metric; Basili und Weiss (1984))

Nicht das messen, was einfach zu bekommen ist,
sondern das, was benötigt wird

1. Ziele erfassen
2. zur Prüfung der Zielerreichung notwendige Fragen ableiten
3. was muss gemessen werden, um diese Fragen zu beantworten

Zielorientiertes Messen



Beispiel: Prozess

Ziel	Frage	Metrik
Maximiere Kundenzufriedenheit	Wie viele Probleme treten beim Kunden auf?	• #Fehler (FR) und #Änderungswünsche (ÄR) • Zuverlässigkeit • Break/Fix-Verhältnis
	Wie lange dauert Problembehebung?	• Verhältnis und Dauer offener und geschlossener FR/ÄR
	Wo sind Flaschenhälse?	• Personalnutzung • Nutzung anderer Ressourcen

Beispiel: Produkt

Ziel	Frage	Metrik
Maximiere Verständlichkeit des Codes	Wie groß ist das System?	• Anzahl Funktionen, Klassen, Pakete etc. • Lines-of-Code pro Funktion, Klasse, Paket etc.
	Wie komplex ist das System?	• McCabe-Komplexität pro Funktion, Klasse, Paket etc. • Schachtelungstiefe pro Funktion • Kopplung • Anzahl Funktionsaufrufe

Wiederholungs- und Vertiefungsfragen

- Was ist ein Maß? Was ist eine Metrik?
- Was ist eine Software-Metrik?
- Welche Skalen gibt es für Daten? Welche Eigenschaften haben diese?
- Beschreiben Sie das Vorgehen bei der Definition und Einführung eines Maßes. Was unterscheidet die interne von der externen Validierung?
- Wie lassen sich Software-Metriken klassifizieren? Nennen Sie Beispiele für jede Klasse.
- Was ist die Bedeutung von Metriken im Software-Entwicklungsprozess?
- Was ist die GQM-Methode? Erläutern Sie GQM anhand des Zieles X.

N.B.: Die Übungsaufgaben sind weitere Beispiele relevanter Fragen.

Kosten- und Aufwandsschätzung

Kostenschätzung

Wichtige Fragen vor einer Software-Entwicklung:

- Wie hoch wird der Aufwand sein?
- Wie lange wird die Entwicklung dauern?
- Wie viele Leute werden benötigt?

Frühzeitige Beantwortung wichtig für:

- Kalkulation und Angebot
- Personalplanung
- Entscheidung „make or buy“
- Nachkalkulation

Kostenschätzung

Ansätze:

- Expertenschätzung
- Berechnung aus früh bekannten Größen (algorithmische Schätzung)
  - COCOMO: Anzahl Codezeilen
  - Function Points: Ein- und Ausgaben
- Analogiemethode
- Top-Down-Schätzung: Ableitung aus globalen Größen
  - z.B. Aufwand steht fest, daraus Umfang ableiten
- Bottom-Up-Schätzung: Dekomposition und Schätzung der einzelnen Komponenten sowie deren Integrationsaufwand
- Daumen-Regeln
  - Gesamtaufwand: 1 DLOC/h (Delivered Line of Code)
  - Gesamtkosten: 50 Euro/DLOC
- Pricing-to-Win
- Parkinsons Gesetz

Kostenschätzung

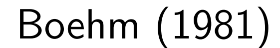
Ansätze:

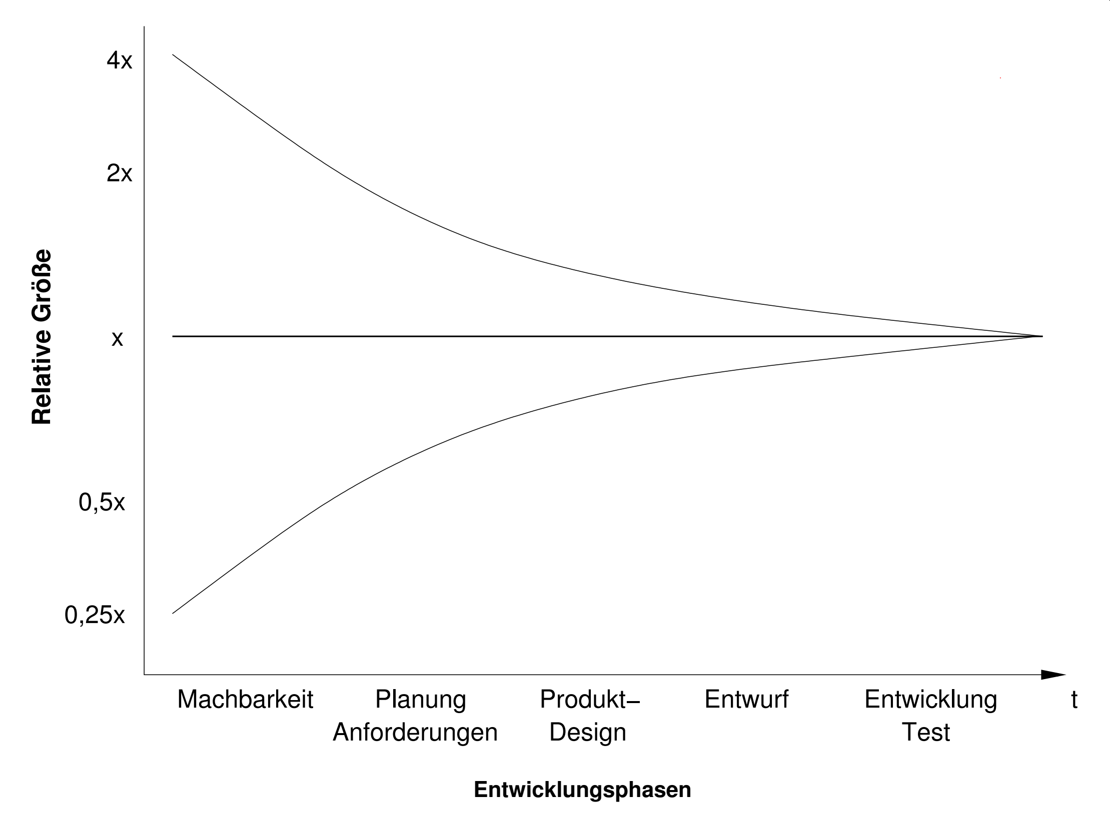
- Expertenschätzung
- Berechnung aus früh bekannten Größen (algorithmische Schätzung)
  - COCOMO: Anzahl Codezeilen
  - Function Points: Ein- und Ausgaben
- Analogiemethode
- Top-Down-Schätzung: Ableitung aus globalen Größen
  - z.B. Aufwand steht fest, daraus Umfang ableiten
- Bottom-Up-Schätzung: Dekomposition und Schätzung der einzelnen Komponenten sowie deren Integrationsaufwand
- Daumen-Regeln
  - Gesamtaufwand: 1 DLOC/h (Delivered Line of Code)
  - Gesamtkosten: 50 Euro/DLOC
- Pricing-to-Win
- Parkinsons Gesetz

- mehrere Experten fragen; Abweichungen diskutieren, bis Konsens erreicht
- statistisches Kostenmodell aus Vergangenheitsdaten wird erstellt; Modell wird zur Vorhersage benutzt
- Bezug auf historische Daten eines ähnlichen Projekts
- ...
- Pricing-To-Win: Preis wird vereinbart; im Zuge des Projekts einigt man sich auf Funktionsumfang
- Parkinsons Gesetz: wenn X Zeit zur Verfügung steht, wird X Zeit benötigt

Interessanterweise sind Manager gar nicht so schlecht im Schätzen von Aufwänden; schlecht sind sie jedoch in der Schätzung von LOC.

Kostenschätzung





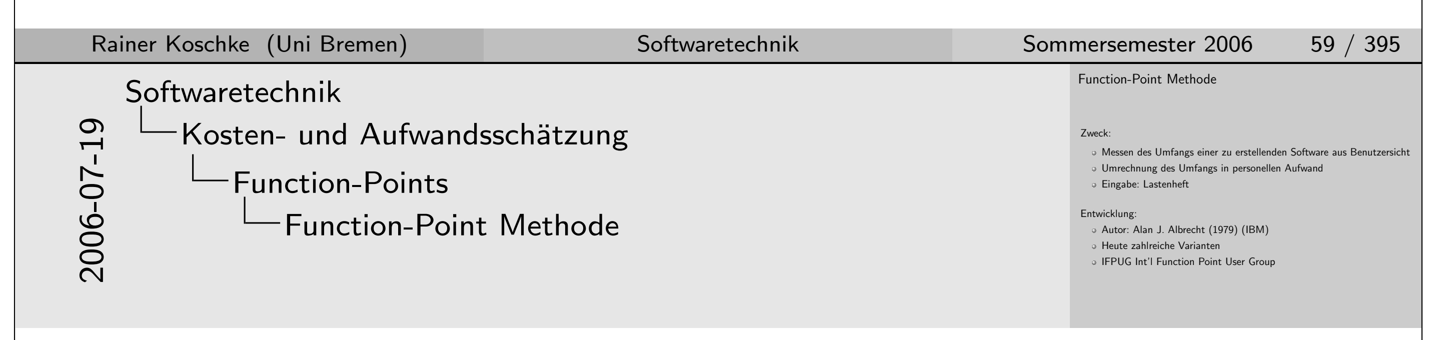
Function-Point Methode

Zweck:

- Messen des Umfangs einer zu erstellenden Software aus Benutzersicht
- Umrechnung des Umfangs in personellen Aufwand
- Eingabe: Lastenheft

Entwicklung:

- Autor: Alan J. Albrecht (1979) (IBM)
- Heute zahlreiche Varianten
- IFPUG Int'l Function Point User Group



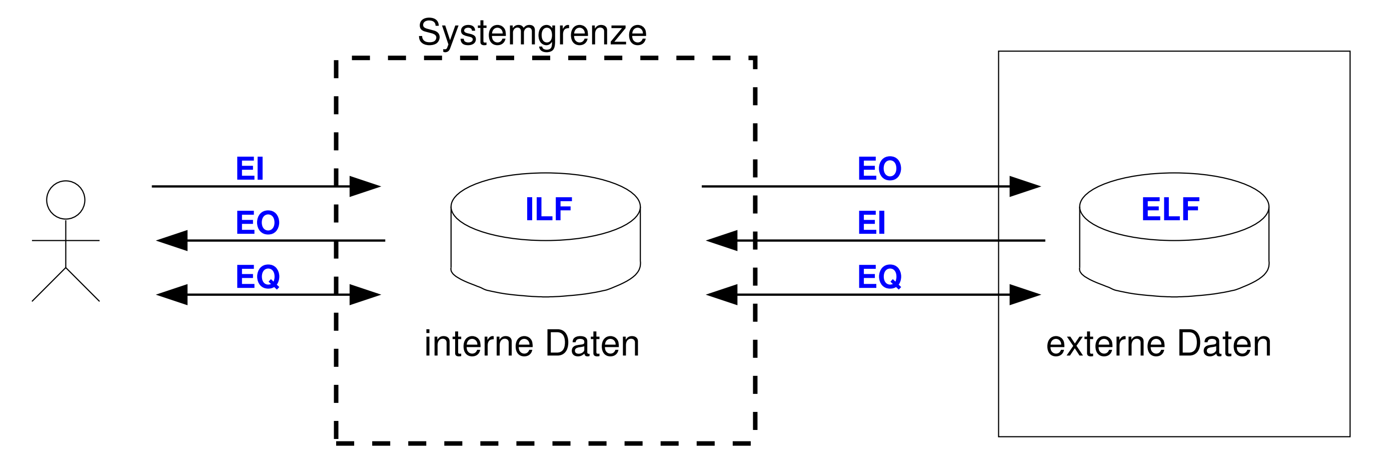
http://www.ifpug.com/fpafund.htm

FAQ: http://ourworld.compuserve.com/homepages/softcomp/fpfaq.htm

Function-Point Methode – Vorgehen

- Zähltyp festlegen: Neu-/Weiterentwicklung, bestehendes System
- Systemgrenzen festlegen
- Identifizieren der Funktionstypen
- Bewerten der Komplexität der Funktionstypen
- Ermittlung der gewichteten Function Points
- Ermittlung des Aufwands

FP – Identifizieren von Funktionstypen



Transaktions-Funktionstypen:

- Eingaben: EI (External Input; Eingabe für ILF)
- Ausgaben: EO (External Output; Ausgabe abgeleiteter Daten)
- Abfragen: EQ (External Inquiry; Eingabe: Anfrage, Ausgabe: Daten)

Daten-Funktionstypen:

- Interner Datenbestand: ILF (Internal Logical File)
- Externer Datenbestand: ELF (External Logical File)

FP – Identifizieren von Funktionstypen

Schlüsselwörter geben Hinweise:

- EI: ablegen/speichern, de-/aktivieren, abbrechen, ändern/editieren/modifizieren/ersetzen, einfügen/hinzufügen, entfernen/löschen, erstellen, konvertieren, update, übertragen
- EO: anzeigen, ausgeben, ansehen, abfragen, suchen/durchsuchen, darstellen, drucken, selektieren, Anfrage, Abfrage, Report
- EQ: abfragen, anzeigen, auswählen, drucken, suchen/durchsuchen, darstellen/zeigen, drop down, extrahieren, finden, holen, selektieren, Ausgabe, Liste, Report

Online-Bibliographie: Startseite



Online-Bibliographie: Taxonomiesuche

Bibliographic References: Taxonomy - Mozilla

File Edit View Go Bookmarks Tools Window Help

http://www.iste.uni-stuttgart.de/ps/reengineering/taxonomy.html Search

Home Bookmarks The Mozilla Org... Latest Builds

Home **Next:** Terminology **Next:** Description

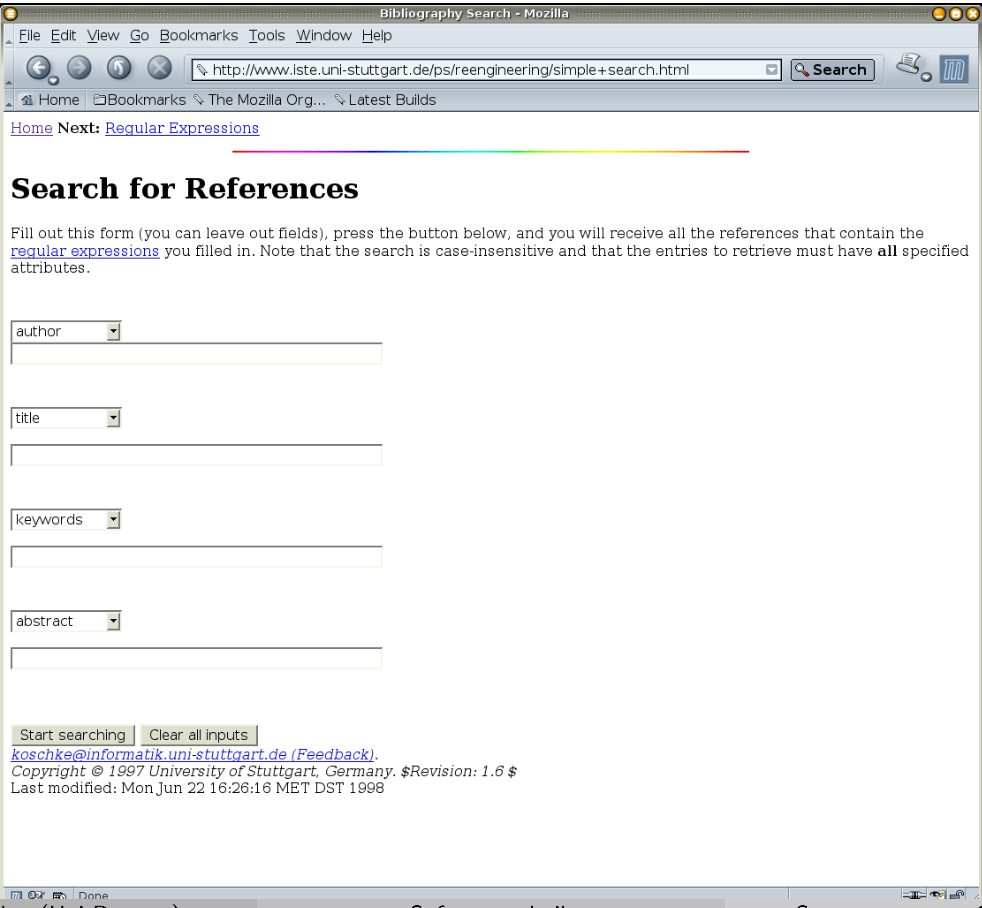
Taxonomy

Select elements of the following taxonomy to retrieve their references or one of the following two items:

- Retrieve used terminology
- Get a description of this taxonomy.

- Reengineering in General
  - Reengineering Collections
  - Introductions to Reengineering
  - Fundamentals
  - The Pros and Cons and Risks of Reengineering
  - Experiences
  - Costs
  - Process Models
  - Management
  - Legality
  - Interoperability
  - Business Reengineering
- Software Reverse Engineering
  - General Information on Reverse Engineering
  - Reverse Engineering Collections
  - Cognitive Processes in Human Program Understanding
  - Software Evolution
  - Extracting Business Rules
  - Intermediate Representations of Source Code
    - Use of data bases
    - Using graphs
  - Preventive Measures
  - Formal Methods
  - Reverse Specification
    - Model Generating
    - Software Animation
      - Visualization of Parallel and Distributed Programs

Online-Bibliographie: Suche nach Eigenschaften



Beispiel: Online-Bibliographie

Systemgrenzen der Online-Bibliographie

- EI_1: BibTeX-Datei importieren
- EO_1: Abfrage über Taxonomie anzeigen lassen
- EQ_1: Artikel über Suchmaske abfragen
- ILF_1: Referenzen-Datenbank
- ELF_1: externe BibTeX-Datei

Function Points zusammenfassen

Parameter	Zähler	Gewicht	Wert
EI	c_1	w_1	$v_1 = c_1 \times w_1$
EO	c_2	w_2	$v_2 = c_2 \times w_2$
EQ	c_3	w_3	$v_3 = c_3 \times w_3$
ILF	c_4	w_4	$v_4 = c_4 \times w_4$
ELF	c_5	w_5	$v_5 = c_5 \times w_5$
Unadjusted Function Points (UFP)			$\sum v_i$

- zu schätzen: c_i und w_i
- feinere Aufteilung ist möglich
  - z.B. 3 EI mit Gewicht 6 und 4 EI mit Gewicht 3

Umfang $\sim$ Summe FPs; „ungewichtete Funktionspunkte“ (UFP)

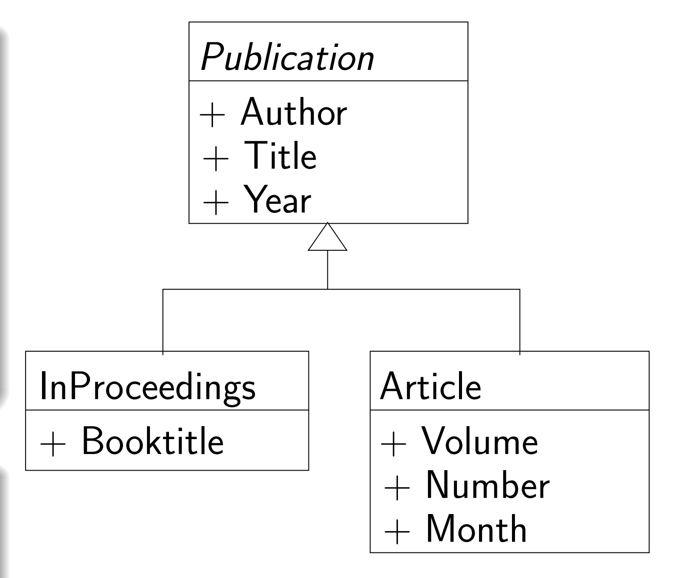
Beispiel: Komplexitätsgewichte w_i für ELF und ILF

Definition

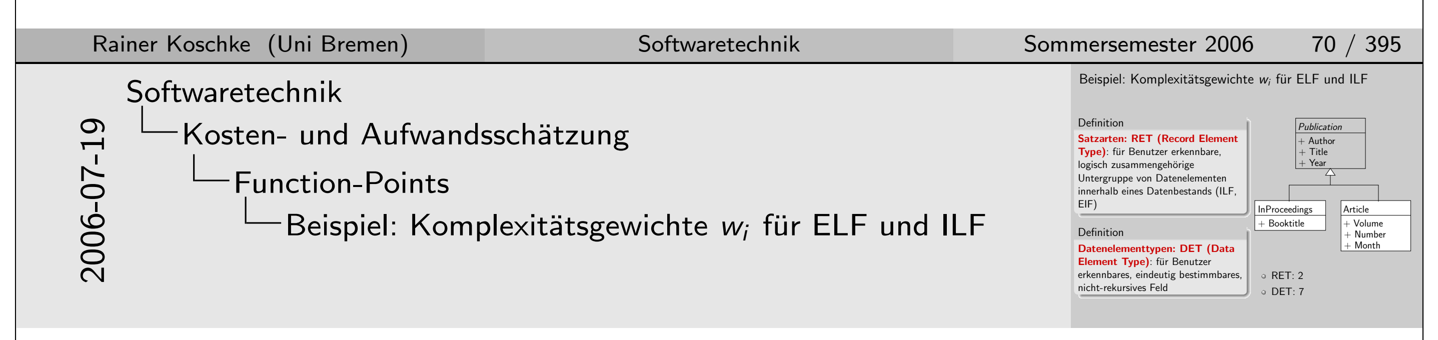
Satzarten: RET (Record Element Type): für Benutzer erkennbare, logisch zusammengehörige Untergruppe von Datenelementen innerhalb eines Datenbestands (ILF, EIF)

Definition

Datenelementtypen: DET (Data Element Type): für Benutzer erkennbares, eindeutig bestimmbares, nicht-rekursives Feld



- RET: 2
- DET: 7



Hier werden die Daten abgeschätzt.

Function Points zusammenfassen

Parameter	Zähler	RET	DET	Gewicht	Wert
ILF_1	1	2	7		
ELF_1	1	2	7		
Parameter	Zähler	FTR	DET	Gewicht	Wert
EI_1					
EO_1					
EQ_1					
Unadjusted Function Points (UFP)					

Komplexitätsgewichte w_i für EI, EO, EQ

Definition

Referenzierte Datenbestände: FTR (File Type Referenced): von Transaktion verwendeter Datenbestand (ILF, ELF)

Beispiel: Datenbank oder Textdatei, die bei der Ausgabe von Kundendaten herangezogen wird

Beispiele für DETs im Kontext Transaktionen:
Eingabe-/Ausgabefelder (GUI), Spalten u.Ä. bei Berichten

Komplexitätsgewichte w_i für EI, EO, EQ

Definition
Referenzierte Datenbestände: FTR (File Type Referenced): von Transaktion verwendeter Datenbestand (ILF, ELF)

Beispiel: Datenbank oder Textdatei, die bei der Ausgabe von Kundendaten herangezogen wird

Beispiele für DETs im Kontext Transaktionen:
Eingabe-/Ausgabefelder (GUI), Spalten u.Ä. bei Berichten

Hier werden die Funktionen abgeschätzt.

Function Points zusammenfassen

- EI_1: BibTeX-Datei importieren
- EO_1: Abfrage über Taxonomie anzeigen lassen
- EQ_1: Artikel über Suchmaske abfragen
- ILF_1: Referenzen-Datenbank
- ELF_1: externe BibTeX-Datei

Parameter	Zähler	RET	DET	Gewicht	Wert
ILF_1	1	2	7		
ELF_1	1	2	7		
Parameter	Zähler	FTR	DET	Gewicht	Wert
EI_1	1	2	7		
EO_1	1	1	7		
EQ_1	1	1	7*		
Unadjusted Function Points (UFP)					

*) jedes DET wird nur einmal gezählt

Function Points zusammenfassen

- EI_1: BibTeX-Datei importieren
- EO_1: Abfrage über Taxonomie anzeigen lassen
- EQ_1: Artikel über Suchmaske abfragen
- ILF_1: Referenzen-Datenbank
- ELF_1: externe BibTeX-Datei

Parameter	Zähler	RET	DET	Gewicht	Wert
ILF_1	1	2	7		
ELF_1	1	2	7		
Parameter	**Zähler**	**FTR**	**DET**	**Gewicht**	**Wert**
EI_1	1	2	7		
EO_1	1	1	7		
EQ_1	1	1	7*		
Unadjusted Function Points (UFP)					

*) jedes DET wird nur einmal gezählt

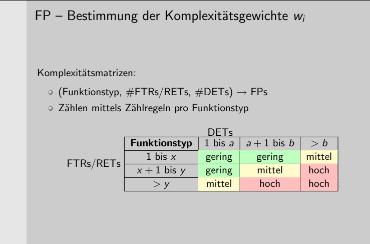
Bei EI_1 werden 7 DETs von ELF_1 gelesen und 7 DETs von ILF_1 geschrieben. Die geschriebenen DETs kreuzen aber nicht die Systemgrenze, so dass insgesamt nur 7 DETs und nicht 14 betrachtet werden.
Bei EQ_1 werden 4 DETs der Suchmaske gelesen (Author, Title, Keywords und Abstract) und 7 DETs von ILF_1 ausgegegen. Aus logischer Sicht werden aber die gleichen DETs (Author, Title, Keywords und Abstract) gelesen und angezeigt, so dass insgesamt nur 7 DETs und nicht 11 betrachtet werden.

FP – Bestimmung der Komplexitätsgewichte w_i

Komplexitätsmatrizen:

- (Funktionstyp, #FTRs/RETs, #DETs) $\rightarrow$ FPs
- Zählen mittels Zählregeln pro Funktionstyp

		DETs		
	Funktionstyp	1 bis a	$a+1$ bis b	$> b$
FTRs/RETs	1 bis x	gering	gering	mittel
	$x+1$ bis y	gering	mittel	hoch
	$> y$	mittel	hoch	hoch


FP – Bestimmung der Komplexitätsgewichte w_i

Komplexitätsmatrizen:
- (Funktionstyp, #FTRs/RETs, #DETs) → FPs
- Zählen mittels Zählregeln pro Funktionstyp

	Funktionstyp	DETs 1 bis a	$a+1$ bis b	$> b$
FTRs/RETs	1 bis x	gering	gering	mittel
	$x+1$ bis y	gering	mittel	hoch
	$> y$	mittel	hoch	hoch

FTR: number of files updated or referenced. A record element type is a user recognizable subgroup of data elements within an ILF or EIF. A data element type is a unique user recognizable, nonrecursive, field.

Zählen von Datenelementtypen (DET)

Ein Datenelementtyp (DET) ist aus der Benutzersicht eindeutig bestimmbares, nicht rekursives Feld, das von der zu bewertenden externen Eingabe (EI) in einem internen Datenbestand (ILF) gepflegt wird.

Zählen Sie 1 DET für jedes aus Benutzersicht nicht rekursive Feld, das die Systemgrenze kreuzt und gebraucht wird, um den Elementarprozess abzuschließen.

Beispiel: Ein Texteingabefeld, in dem der Nachname eines neuen Kunden eingegeben wird, wird als 1 DET gezählt.

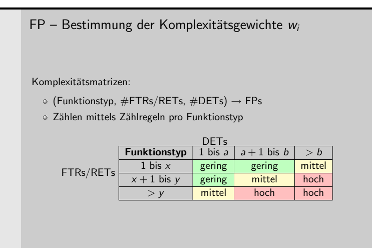
Gegenbeispiel: Eine Dateiauswahlliste, in der beliebig viele Dateien von der Festplatte des Benutzers ausgewählt werden können, ist rekursiv, und muss somit gesondert gezählt werden.

Zählen Sie keine Felder, die durch das System gesucht und/oder in einem ILF gespeichert werden, wenn die Daten nicht die Systemgrenze überqueren.

Zählen Sie nur 1 DET für die Fähigkeit, eine Systemantwort als Meldung für einen aufgetretenen Fehler aus der Anwendung heraus zu senden bzw. für die Bestätigung, dass die Verarbeitung beendet oder fortgesetzt werden kann Beispiel: Bei Eingabe eines Datums soll z.B. das Format TT/MM/JJJJ eingehalten werden. Gibt der Bearbeiter z.B. '12.03.1997' ein und bestätigt seine Eingabe, so erhält er die Meldung 'neuer Datensatz gespeichert'. Gibt er hingegen '12.3.97' ein (Jahreszahl nicht vierstellig) so erhält er die Fehlermeldung 'Fehler: Bitte Datum korrigieren'. Nur ein DET wird für diese Fähigkeit des Systems gezählt.

Zählen Sie nur 1 DET für die Möglichkeit, eine Aktion durchzuführen, auch wenn es viele Methoden gibt, die denselben logischen Prozess anstoßen. Beispiel: In einem Eingabeformular gibt es einen OK-Button zum Absenden der Daten. Die Tastaturkombination STRG-S führt ebenfalls zum Senden der Daten. Somit wird nur ein DET gezählt.

2006-07-19
Softwaretechnik
Kosten- und Aufwandsschätzung
Function-Points
FP – Bestimmung der Komplexitätsgewichte w_i


FP – Bestimmung der Komplexitätsgewichte w_i

Komplexitätsmatrizen:
- (Funktionstyp, #FTRs/RETs, #DETs) → FPs
- Zählen mittels Zählregeln pro Funktionstyp

	Funktionstyp	DETs 1 bis a	$a+1$ bis b	$> b$
FTRs/RETs	1 bis x	gering	gering	mittel
	$x+1$ bis y	gering	mittel	hoch
	$> y$	mittel	hoch	hoch

(Fortsetzung)

Zählen von referenzierte Datenbeständen (FTR)

Ein referenzierter Datenbestand (FTR) ist eine vom Benutzer definierte Gruppierung zusammengehöriger Daten oder Steuerungsinformationen in einem internen Datenbestand (ILF), die bei der Bearbeitung der externen Eingabe gelesen oder gepflegt wird.

Zählen Sie 1 FTR für jeden referenzierten Datenbestand, der während der Verarbeitung der externen Eingabe gelesen wird. Beispiel: Es werden durch eine externe Eingabe Produktdaten in einer Datenbank gespeichert. Dazu werden die Produktbezeichnungen aus einer weiteren Datenbank ausgelesen, die damit zusätzlich zu der zu aktualisierenden Produktdatenbank einen weiteren Datenbestand darstellt, der jedoch nur gelesen wird.

Zählen Sie 1 FTR für jede ILF, die während der Verarbeitung der externen Eingabe gepflegt wird. Beispiel: Es wird zusätzlich zu den Aktionen des vorigen Beispiels eine Textdatei aktualisiert, in der die Anzahl der Zugriffe auf die Datenbank verzeichnet wird.

Zählen Sie nur 1 FTR für jede ILF, die während der Verabeitung der externen Eingabe gelesen und gepflegt wird. Beispiel: Würden die Informationen der Textdatei ebenfalls in der Datenbank gespeichert werden, so wird diese nur als 1 FTR gezählt, obwohl die Datenbank zur Ein- und Ausgabe von Daten verwendet wird.

FP – Bestimmung der Komplexitätsgewichte w_i

Komplexitätsmatrizen:

- (Funktionstyp, #FTRs/RETs, #DETs) → FPs
- Zählen mittels Zählregeln pro Funktionstyp

FTRs/RETs	**Funktionstyp**	DETs: 1 bis a	$a+1$ bis b	$> b$
	1 bis x	gering	gering	mittel
	$x+1$ bis y	gering	mittel	hoch
	$> y$	mittel	hoch	hoch

(Fortsetzung)
Besonderheiten bei grafischen Benutzungsoberflächen
Besonderheiten bei grafischen Benutzungsoberflächen Optionsfelder (Radiobuttons) stellen Datenelemente dar. Es wird pro Gruppe von Optionsfeldern 1 DET gezählt, da innerhalb einer Gruppe nur ein Optionsfeld ausgewählt werden kann. Beispiel: Eine Gruppe von 12 Radiobuttons, in der ein PKW-Typ ausgewählt werden kann, wird als 1 DET gezählt.
Kontrollkästchen (Checkboxen) stellen Datenelemente dar. Im Gegensatz zu Optionsfeldern können aus einer Gruppe von Checkboxen mehrere Elemente gleichzeitig ausgewählt werden. Somit wird jedes Element als 1 DET gezählt. Beispiel: Eine Gruppe von 12 Checkboxen, mit der ein Pizza-Belag zusammengestellt werden kann, wird als 12 DETs gezählt.
Eingabe- und Ausgabefelder stellen Datenelemente dar. Beispiel: In einer Bildschirmmaske werden Vorname, Nachname, Straße, Hausnummer, PLZ und Ort in Eingabefeldern erfasst. Somit werden 6 DET gezählt.
Literale stellen keine Datenelemente dar. Beispiel: Vor einem Feld ist die Textzeile 'monatliches Gehalt' und dahinter 'in Euro/Monat' angegeben. Beide Textzeilen sind Literale und werden nicht gezählt
Enter-, OK-Button und Programmfunktionstaste werden insgesamt als 1 DET gezählt, da jeweils die gleiche Funktion ausgeführt wird. Beispiel: Die Daten eines Dialogs werden nach Betätigen der Enter-Taste oder nach Betätigen der Schaltfläche 'übernehmen' (gleiche Funktion wie OK-Button) in einer Datenbank gespeichert. Es wird 1 DET gezählt.
Berichte/Reports können verschiedene Ausgabeformen haben. So kann die gleiche Datenbasis zur Darstellung als Tortendiagramm, Tabelle, textuell, als druckbares Format oder als Exportdatei dargestellt werden. Jedes Format stellt dabei eine externe Ausgabe (EO) dar.

FP – Werte der Komplexitätsgewichte w_i

FTRs	**EI**	DETs: 1-4	5-15	16+
	0-1	3	3	4
	2	3	4	6
	3+	4	6	6

FTRs	**EO**	DETs: 1-5	6-19	20+
	0-1	4	4	5
	2-3	4	5	7
	4+	5	7	7

FTRs	**EQ**	DETs: 1-5	6-19	20+
	0-1	3	3	4
	2-3	3	4	6
	4+	4	6	6

RETs	**ILF**	DETs: 1-19	20-50	51+
	1	7	7	10
	2-5	7	10	15
	6+	10	15	15

RETs	**ELF**	DETs: 1-19	20-50	51+
	1	5	5	7
	2-5	5	7	10
	6+	7	10	10

Beispiel: Function Points zusammenfassen

- EI_1: BibTeX-Datei importieren
- EO_1: Abfrage über Taxonomie anzeigen lassen
- EQ_1: Artikel über Suchmaske abfragen
- ILF_1: Referenzen-Datenbank
- ELF_1: externe BibTeX-Datei

Parameter	Zähler	RET	DET	Gewicht	Wert
ILF_1	1	2	7	7	7
ELF_1	1	2	7	5	5
Parameter	Zähler	FTR	DET	Gewicht	Wert
EI_1	1	2	7	4	4
EO_1	1	1	7	4	4
EQ_1	1	1	7	3	3
Unadjusted Function Points (UFP)					23

FP – Gewichtete Function-Points

Systemmerkmale:

- Datenkommunikation
- Verteilte Verarbeitung
- Leistungsfähigkeit
- Begrenzte Kapazität
- Transaktionsrate
- Interaktive Dateneingabe
- Benutzerfreundlichkeit
- Interaktive Änderung
- Komplexe Verarbeitung
- Wiederverwendbarkeit
- Installationshilfen
- Betriebshilfen
- Mehrfachinstallation
- Änderungsfreundlichkeit

Bewertung: 0 = kein Einfluss, 5 = starker Einfluss

Konkrete Fragen I

- Does the system require reliable backup and recovery? **3**
- Are data communications required? **2**
- Are there distributed processing functions? **0**
- Is performance critical? **1**
- Will the system run in an existing, heavily utilized operational environment? **1**
- Does the system require on-line data entry? **4**
- Does the online data entry require the input transaction to be built over multiple screens or operations? **3**

Konkrete Fragen II

- Are the master files updated on-line? **5**
- Are the inputs, outputs, files, or inquiries complex? **1**
- Is the internal processing complex? **1**
- Is the code designed to be reusable? **1**
- Are conversion and installation included in the design? **2**
- Is the system designed for multiple installations in different organizations? **2**
- Is the application designed to facilitate change and ease of use by the user? **3**

Konkrete Fragen II

- Are the master files updated on-line? 5
- Are the inputs, outputs, files, or inquiries complex? 1
- Is the internal processing complex? 1
- Is the code designed to be reusable? 1
- Are conversion and installation included in the design? 2
- Is the system designed for multiple installations in different organizations? 2
- Is the application designed to facilitate change and ease of use by the user? 3

Die Zahlen beziehen sich auf unser laufendes Beispiel.

FP – Gewichtete Function-Points

- TDI (Total Degree of Influence) = Summe der Bewertungen
- VAF (Value Adjustment Factor) = TDI/100 + 0,65
  → Gesamteinflussfaktor: 65% - 135%
- AFP (Adjusted Function Points) = UFP · VAF

Beispiel:

- TDI = 29
- VAF = 29/100 + 0,65 = 0,94
- AFP = 23 · 0,94 = 21,62

FP – Umrechnung in Aufwand

Gesucht: Abbildung FPs $\rightarrow$ Aufwand

- Erstellung einer neuen Erfahrungskurve
  (Zählen abgeschlossener Projekte, Regressionsanalyse)
- Datenbank, z.B. ISBSG[1]:
  - 3GL-Projekte: $PM = 0,971 \cdot AFP^{0,351}$
  - 4GL-Projekte: $PM = 0,622 \cdot AFP^{0,405}$
  - basierend auf 662 Projekten: $PM = 0,38 \cdot AFP^{0,37}$
- grobe Schätzung mit Faustregeln Jones (1996):
  - Entwicklungsdauer (Monate) $= AFP^{0.4}$
  - Personen = AFP / 150 (aufgerundet)
  - Aufwand (Personenmonate) = Personen $\cdot$ Entwicklungsdauer

Beispiel (mit Jones-Schätzung):

- Entwicklungsdauer (Monate) $= 21.62^{0.4} = 3.42$
- Personen $= 21.62/150 \rightarrow 1$
- Aufwand (Personenmonate) $= 1 \cdot 3.42 = 3.42$

[1] International Software Benchmarking Standards Group

FP – Umrechnung in Aufwand
Gesucht: Abbildung FPs → Aufwand
Erstellung einer neuen Erfahrungskurve
(Zählen abgeschlossener Projekte, Regressionsanalyse)
Datenbank, z.B. ISBSG1:
3GL-Projekte: PM = 0,971 · AFP0,351
4GL-Projekte: PM = 0,622 · AFP0,405
basierend auf 662 Projekten: PM = 0,38 · AFP0,37
grobe Schätzung mit Faustregeln Jones (1996):
Entwicklungsdauer (Monate) = AFP0.4
Personen = AFP / 150 (aufgerundet)
Aufwand (Personenmonate) = Personen · Entwicklungsdauer
Beispiel (mit Jones-Schätzung):
Entwicklungsdauer (Monate) = 21.620.4 = 3.42
Personen = 21.62/150 → 1
Aufwand (Personenmonate) = 1 · 3.42 = 3.42
1International Software Benchmarking Standards Group

`http://www.isbsg.org/ http://www.isbsg.org/isbsg.nsf/weben/Project%20Duration`

FP – Umrechnung in LOC

Mittlere Anzahl Codezeilen pro FP (Jones 1995):

Sprache	ØLOC
Assembler	320
C	128
FORTRAN	107
COBOL (ANSI 85)	91
Pascal	91
C++	53
Java	53
Ada 95	49
Smalltalk	21
SQL	12

Bewertung der Function-Point-Methode

+ Wird als beste verfügbare Methode zur Schätzung kommerzieller Anwendungssysteme angesehen (Balzert 1997)
+ Sinnvoll einsetzbar, wenn Erfahrungswerte von vergleichbaren Projekten vorliegen (Kemerer 1987)
– Bewertung der Systemmerkmale subjektiv (Symons 1988)
+ Studie: mittlere FP-Abweichung zwischen 2 „Zählern“ nur 12% (Kemerer und Porter 1992)
– Zählen der FPs relativ aufwendig

Object Points

- Für 4GLs (Query Languages, Report Writers, ...)
- Haben nicht unbedingt mit OOP-Objekten zu tun
- Gewichtete Schätzung von
  - Anzahl verschiedener „Screens“
  - Anzahl erstellter „Reports“
  - Anzahl zu entwickelnder 3GL-Module
- Vorteil: Einfacher und weniger zu schätzen:
  vergleichbare Präzision wie Function-Point-Schätzung (Banker u. a. 1991)
  47% des Aufwands für Function-Point-Schätzung (Kauffman und Kumar 1993)

Object Points

Screens

# views contained	# data tables < 4	< 8	$8+$
< 3	1	1	2
3-7	1	2	3
> 8	2	3	3

Reports

# sections contained	# data tables < 4	< 8	$8+$
0-1	2	2	5
2-3	2	5	8
4+	5	8	8

Views: Menge logisch zusammengehöriger Daten (z.B. Kundenstammdaten)

Data Tables = # Server data tables + # Client data tables (Tabellen, die abgefragt werden müssen, um Daten zu bestimmen)

Jede 3GL-Komponente: 10 object points

COCOMO

COCOMO = Constructive Cost Model (Boehm 1981)

- Basiert auf Auswertung sehr vieler Projekte
- Eingaben: Projektkomplexität (3 Stufen), Systemgröße
- Ausgaben: Realisierungsaufwand, Entwicklungszeit
- Drei Genauigkeitsstufen (steigender Aufwand):
  - *Basic:* $\text{Aufwand} = a \cdot \text{KLOC}^b$, $\text{Dauer} = c \cdot \text{Aufwand}^d$
  - *Intermediate:* Dekomposition, 15 Einflussfaktoren (Kategorien: Produkt, Projekt, Computer, Personal)
  - *Advanced:* Einflussfaktoren pro Phase

COCOMO

a, b konstant, abhängig von Projektkomplexität:

- *Organic:* $\text{PM} = 2,4 \cdot \text{KDSI}^{1,05} \cdot M$
  - wohl verstandene Anwendungsdomäne mit kleinen Teams
- *Semidetached:* $\text{PM} = 3,0 \cdot \text{KDSI}^{1,12} \cdot M$
  - komplexere Projekte, bei dem Teams nur begrenzte Erfahrungen haben
- *Embedded:* $\text{PM} = 3,6 \cdot \text{KDSI}^{1,20} \cdot M$
  - Projekte, eingebettet in komplexe Systeme aus Hardware, Software, Vorschriften und betriebliche Abläufe

KDSI = Kilo Delivered Source Instructions

M ergibt sich aus Einflussfaktoren

COCOMO

Damals:

- Wasserfallprozess
- nur Neuentwicklung
- Mainframes

Heute:

- Inkrementelle Entwicklung
- Wiederverwendung, COTS-Komponenten
- PCs
- Reengineering
- Code-Generierung

→ COCOMO II (Boehm u. a. 1995)

COCOMO II

Unterscheidung nach Phasen (Boehm u. a. 2000):

- Frühe Prototypenstufe
- Frühe Entwurfsstufe
- Stufe nach Architekturentwurf

Spätere Schätzung → höhere Genauigkeit

COCOMO II – Early prototyping level

Eingaben:

- Object Points (OP)
- Produktivität (PROD):

Erfahrung/Fähigkeiten der Entwickler	- -	-	o	+	++
Reife/Fähigkeiten der CASE-Tools	- -	-	o	+	++
PROD (NOP/Monat)	4	7	13	25	50

- Wiederverwendungsanteil *%reuse* in Prozent

Abgeleitete Größen:

- New Object Points (NOP): berücksichtigen Wiederverwendung
  $NOP = \text{OP} \cdot (100 - \%\text{reuse})/100$
- Aufwand in Personenmonaten PM = NOP/PROD

Unterstützt Prototypen, Wiederverwendung

COCOMO II – Early design level

- Schätzung basiert auf Function Points (LOCs werden daraus abgeleitet)
- Personenmonate $PM_{NS} = A \cdot \text{KLOC}^{E} \cdot EM + \text{PM}_{\text{m}}$ bei nominalem Zeitplan
- $A = 2,94$ in initialer Kalibrierung
- Exponent E:
  - 5 Faktoren w_i für Exponent E (5 = sehr klein, 0 = sehr groß): Erfahrung mit Domäne, Flexibilität des Entwicklungsprozesses, Risikomanagement, Teamzusammenhalt, Prozessreife
  - $E = B + \sum w_i/100$ mit $B = 1.01$
- Effort Multiplier EM:
  - 7 lineare Einflussfaktoren (6 Stufen, Standard: 1.00, in Tabelle nachschlagen): Produktgüte und -komplexität, Plattformkomplexität, Fähigkeiten des Personals, Erfahrung des Personals, Zeitplan, Infrastruktur
  - $EM = \prod \textit{Effort-Multiplier}_{\text{i}}$
- Korrekturfaktor PM_{m} bei viel generiertem Code (höhere Produktivität; nicht weiter diskutiert hier)

Faktoren für Exponent E I

- Erfahrung mit Anwendungsbereich (PREC)
  - → Erfahrung mit vorliegendem Projekttyp
  - 5 keine Erfahrung
  - 0 vollständige Vertrautheit
- Entwicklungsflexibilität (FLEX)
  - → Grad der Flexibilität im Entwicklungsprozess
  - 5 Prozess vom Kunden fest vorgegeben
  - 0 Kunde legt nur Ziele fest
- Risikomanagement (RESL)
  - → Umfang der durchgeführten Risikoanalyse
  - 5 keine Risikoanalyse
  - 0 vollständige und genaue Risikoanalyse

Faktoren für Exponent E II

- Teamzusammenhalt (TEAM)
  - → Vertrautheit und Eingespieltheit des Entwicklungsteams
  - 5 schwierige Interaktionen
  - 0 integriertes und effektives Team ohne Kommunikationsprobleme
- Prozessreife (EPML)
  - → Reife des Entwicklungsprozesses (z.B. CMM);
  - → beabsichtigt: gewichteter Anteil der mit „ja“ beantworteten Fragen im CMM-Fragebogen
  - → pragmatisch: CMM-Level
  - 5 niedrigster CMM-Level
  - 0 höchster CMM-Level

Effort Multiplier RCPX: Product Reliability and Complexity

RELY	Required reliability
DOCU	Documentation match to life-cycle needs
CPLX	Product Complexity
DATA	Data base size

Grad:	very low	low	nominal	high	very high	extra high
Punkte:	1	2	3	4	5	6
RCPX						
RELY	very little	little	some	basic	strong	
DOCU	very little	little	some	basic	strong	
CPLX	very little	little	some	basic	strong	very strong
DATA		small	moderate	large	very large	

$\sum$ Punkte:	5,6	7,8	9–11	12	13–15	16–18	19–21
EM_{RPCX}	0.49	0.60	0.83	1.00	1.33	1.91	2.72

Effort Multiplier RCPX: Product Reliability and Complexity

RELY Required reliability
DOCU Documentation match to life-cycle needs
CPLX Product Complexity
DATA Data base size

Grad: very low low nominal high very high extra high
Punkte: 1 2 3 4 5 6
RCPX
RELY very little little some basic strong
DOCU very little little some basic strong
CPLX very little little some basic strong very strong
DATA small moderate large very large

∑ Punkte: 5,6 7,8 9–11 12 13–15 16–18 19–21
EMRPCX 0.49 0.60 0.83 1.00 1.33 1.91 2.72

DATA: This cost driver attempts to capture the effect large test data requirements have on product development. The rating is determined by calculating D/P, the ratio of bytes in the testing database to SLOC in the program. The reason the size of the database is important to consider is because of the effort required to generate the test data that will be used to exercise the program. In other words, DATA is capturing the effort needed to assemble and maintain the data required to complete test of the program.

Effort Multiplier PDIF: Platform Difficulty

TIME Execution time constraints
STOR Main storage constraints
PVOL Platform volatility

Grad:	low	nominal	high	very high	extra high
Punkte:	2	3	4	5	6
PDIF					
TIME		≤50%	≤65%	≤80%	≤90%
STORE		≤50%	≤65%	≤80%	≤90%
PVOL	very stable	stable	somewhat volatile	volatile	

$\sum$ Punkte:	8	9	10–12	13–15	16,17
EM_{PDIF}	0.87	1.00	1.29	1.81	2.61

Effort Multiplier PERS: Personnel Capability

ACAP Analyst capability (gemessen als Perzentil)
PCAP Programmer capability (gemessen als Perzentil)
PCON Personnel continuity (gemessen durch Personalfluktuation)

Grad:	very low	low	nominal	high	very high
Punkte:	1	2	3	4	5
PERS					
ACAP	15%	35%	55%	75%	90%
PCAP	15%	35%	55%	75%	90%
PCON	48%	24%	12%	6%	3%

$\sum$ Punkte:	3,4	5,6	7,8	9	10,11	12,13	14,15
EM_{PERS}	2.12	1.62	1.26	1.00	0.83	0.63	0.50

Effort Multiplier PERS: Personnel Capability

ACAP Analyst capability (gemessen als Perzentil)
PCAP Programmer capability (gemessen als Perzentil)
PCON Personnel continuity (gemessen durch Personalfluktuation)

Grad:	very low	low	nominal	high	very high
Punkte:	1	2	3	4	5
PERS					
ACAP	15%	35%	55%	75%	90%
PCAP	15%	35%	55%	75%	90%
PCON	48%	24%	12%	6%	3%

$\sum$ Punkte:	3,4	5,6	7,8	9	10,11	12,13	14,15
EM_{PERS}	2.12	1.62	1.26	1.00	0.83	0.63	0.50

A percentile rank is the proportion of scores in a distribution that a specific score is greater than or equal to. For instance, if you received a score of 95 on a math test and this score was greater than or equal to the scores of 88% of the students taking the test, then your percentile rank would be 88. You would be in the 88th percentile.

Effort Multiplier PREX: Personnel Experience

AEXP Applications experience
PLEX Platform experience
LTEX Language/tool experience

Grad:	very low	low	nominal	high	very high
Punkte:	1	2	3	4	5
PREX					
AEXP	≤2 Mo.	6 Mo.	1 J.	3 J.	6 J.
PLEX	≤2 Mo.	6 Mo.	1 J.	3 J.	6 J.
LTEX	≤2 Mo.	6 Mo.	1 J.	3 J.	6 J.

$\sum$ Punkte:	3,4	5,6	7,8	9	10,11	12,13	14,15
EM_{PREX}	1.59	1.33	1.22	1.00	0.87	0.74	0.62

Effort Multiplier FCIL: Facilities

TOOL Use of Software Tools
SITE Multisite Development

Grad:	very low	low	nominal	high	very high		
Punkte:	1	2	3	4	5	6	
FCIL							
TOOL	(1)	(2)	(3)	(4)	(5)		→ nächste Folie
SITE	(1)	(2)	(3)	(4)	(5)	(6)	→ nächste Folie

$\sum$ Punkte:	2	3	4,5	6	7,8	9,10	11
EM_{FCIL}	1.43	1.30	1.10	1.00	0.87	0.73	0.62

TOOL und SITE

TOOL:

1. Editor, Compiler, Debugger
2. einfaches CASE-Werkzeug, schlechte Integration
3. Basis-Life-Cycle-Tools moderat integriert
4. weitergehende, reife Life-Cycle-Tools moderat integriert
5. weitergehende, proaktive Life-Cycle-Tools gut integriert mit Prozessen, Methoden und Wiederverwendung

SITE:

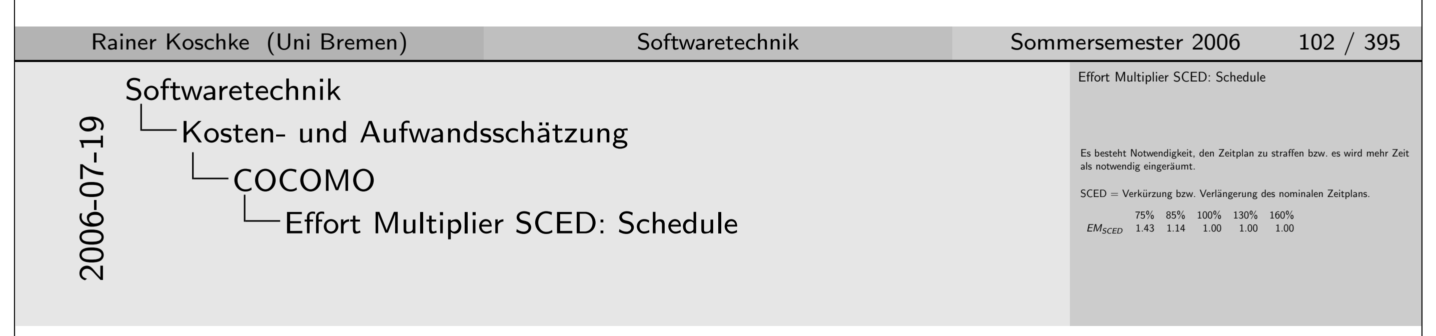
1. Telefon prinzipiell vorhanden und Post
2. individuelles Telefon und Fax
3. E-Mail (niedrige Bandbreite)
4. elektronische Kommunikation mit großer Bandbreite
5. elektronische Kommunikation mit großer Bandbreite, gelegentliche Videokonferenzen
6. Interaktive Multimedia

Effort Multiplier SCED: Schedule

Es besteht Notwendigkeit, den Zeitplan zu straffen bzw. es wird mehr Zeit als notwendig eingeräumt.

SCED = Verkürzung bzw. Verlängerung des nominalen Zeitplans.

	75%	85%	100%	130%	160%
EM_{SCED}	1.43	1.14	1.00	1.00	1.00



This rating measures the schedule constraint imposed on the project team developing the software. The ratings are defined in terms of the percentage of schedule stretch-out or acceleration with respect to a nominal schedule for a project requiring a given amount of effort. Accelerated schedules tend to produce more effort in the later phases of development because more issues are left to be determined due to lack of time to resolve them earlier. A schedule compress of 74% is rated very low. A stretch-out of a schedule produces more effort in the earlier phases of development where there is more time for thorough planning, specification and validation. A stretch-out of 160% is rated very high.
Stretch-outs (i.e., SCED $>$ 100) do not add or decrease effort. Their savings bacause of smaller team size are generally balanced by the need to carry project administrative functions over a longer period of time.

Rechenbeispiel

Nominaler Aufwand

Personenmonate $PM_{NS} = A \cdot \text{KLOC}^E \cdot EM$ mit $EM_{SCED} = 1,0$

mit $A = 2,94$ und $E = B + \sum w_i/100$ mit $B = 1,01$.

Annahme: es herrschen einfache Verhältnisse:

- $\rightarrow$ $\forall i : w_i = 0 \Rightarrow E = 1,01$ (bester Fall)
- $\rightarrow$ nominale Effort-Multiplier = 1,00 (Normalfall) $\Rightarrow EM = 1,00$

Geschätzte Lines-of-Code = 100.000

$\rightarrow PM_{NS} = 2,94 \times 100^{1,01} \times 1,0 = 307,86 \text{ Monate } \approx 25\frac{1}{2} \text{ Jahre}$

Entwicklungsdauer

Nominale Entwicklungsdauer (Kalenderzeit in Monaten)

$TDEV_{NS} = C \times PM_{NS}^{D+0,2\times(E-B)}$

mit $C = 3,67$ und $D = 0,28$.

Beispiel: $TDEV_{NS} = 3,67 \times 307,86^{0,28+0,2\times 0} = 18,26$

Anzahl Entwickler

$N = PM_{NS}/TDEV_{NS}$

Beispiel: $N = 307,86/18,26 \approx 17$

Verkürzte Entwicklungsdauer

Chef: „Wieso 18 Monate? Geht das nicht schneller?“

PM_{NS} geht von $SCED = 1,0$ aus.

Abweichung von der nominalen Entwicklungdauer

$TDEV = TDEV_{NS} \times SCED/100$

Wir verkürzen auf 75%:

$TDEV = 18,26 \times 75/100 = 10,3$ Monate

Chef: „Super!“

Wir setzen $SCED = 75$ in PM-Formel ein.

$PM = 2,94 \times 100^{1,01} \times 1,0 \times 1,43 = 440,23$

Erhöhung des Aufwands: $440,23/307,86 = 1,43$

Chef: „43% mehr Kosten? Seid Ihr wahnsinnig?“

COCOMO II – Post-architecture level

Berücksichtigt:

- Auswirkungen erwarteter Änderungen von Anforderungen
- Ausmaß/Aufwand der möglichen Wiederverwendung
  - Aufwand für Entscheidung, ob Wiederverwendung
  - Aufwand für das Verstehen existierenden Codes
  - Aufwand für Anpassungen
- 17 verfeinerte lineare Einflussfaktoren
- Schätzung basiert auf LOC

COCOMO II – Post-architecture level

Produktgüte/-kompl.	→	Verlässlichkeit, Datenbasisgröße, Komplexität, Dokumentation
Plattformkomplexität	→	Laufzeit-, Speicherbeschränkungen, Plattformdynamik
Fähigkeiten Personal	→	Fähigkeiten der Analysten/Entwickler, Kontinuität des Personals
Erfahrung Personal	→	Domänenerfahrung der Analysten/Entwickler, Erfahrung mit Sprache und Werkzeugen
Infrastruktur	→	Tools, verteilte Entwicklung+Kommunikation

Einflussfaktoren (Cost Drivers) für Cocomo-2

	- -	-	o	+	++	+++
Product Attributes						
RELY – Required reliability	0.82	0.92	1.00	1.10	1.26	
DATA – Data base size		0.90	1.00	1.14	1.28	
CPLX – Product Complexity	0.73	0.87	1.00	1.17	1.34	1.74
RUSE – Required Reusability		0.95	1.00	1.07	1.15	1.24
DOCU – Doc. match to life-cycle needs	0.81	0.91	1.00	1.11	1.23	
Computer Attributes						
TIME – Execution time constr.			1.00	1.11	1.29	1.63
STOR – Main storage constr.			1.00	1.05	1.17	1.46
PVOL – Platform volatility		0.87	1.00	1.15	1.30	

Einflussfaktoren (Cost Drivers) für Cocomo-2

	- -	-	o	+	++	+++
Personell attributes						
ACAP – Analyst capability	1.42	1.19	1.00	0.85	0.71	
PCAP – Programmer capability	1.34	1.15	1.00	0.88	0.76	
AEXP – Applications experience	1.22	1.10	1.00	0.88	0.81	
PLEX – Platform experience	1.19	1.09	1.00	0.91	0.85	
LTEX – Language/tool exp.	1.20	1.09	1.00	0.91	0.84	
Project attributes						
TOOL – Use of software tools	1.17	1.09	1.00	0.90	0.78	
SITE – Multisite development	1.22	1.09	1.00	0.93	0.86	0.80
SCED – Required dev. schedule	1.43	1.14	1.00	1.00	1.00	

Zusammenfassung

- alle Schätzungen basieren auf Erfahrung
- kontinuierlich schätzen
- verschiedene Techniken anwenden

Wiederholungs- und Vertiefungsfragen I

- Welche Möglichkeiten zur Schätzung von Aufwand und Kosten für die Software-Entwicklung gibt es?
- Wann wird geschätzt?
- Erläutern Sie die Function-Point-Methode (am konkreten Beispiel).
- Was sind Adjusted Function-Points im Unterschied zu Unadjusted-Function-Points?
- Wie errechnet sich der Aufwand aus den Function-Points?
- Was ist die Idee von Object-Points im Gegensatz zu Function-Points?
- Erläutern Sie die CoCoMo-Methode (neue Version) für die Level *Early Prototyping* und *Early Design Level*.
- Geben Sie die grundsätzliche Formel für den Aufwand wieder. Was bedeuten die verschiedenen Parameter?
- Um welche Art von Parametern handelt es sich bei den Faktoren, die im Exponenten auftreten?

Wiederholungs- und Vertiefungsfragen II

- Wozu die Unterscheidung in die verschiedene Stufen (Level)?
- Wie errechnet sich die Entwicklungsdauer aus dem Aufwand?
- Wie wird eine Verkürzung der nominalen Entwicklungsdauer behandelt?
- Vergleichen Sie die Function-Point-Methode mit CoCoMo.

N.B.:

1. Die Übungsaufgaben sind weitere Beispiele relevanter Fragen.
2. Bei der Darstellung der Einflussfaktoren für die Adjusted Function Points genügt es, ein paar Beispiele nennen zu können und zu wissen, worauf sie sich grundsätzlich beziehen (System und nicht Entwicklungsteam/-prozess); keinesfalls wird Gedächtnisleistung abgeprüft; das gilt auch für die Einflussfaktoren von und CoCoMo.
3. Bei der Darstellung von Function-Points und CoCoMo interessieren nicht die absoluten Zahlen, aber sehr wohl der grundsätzliche Aufbau der Formeln.

Lernziele

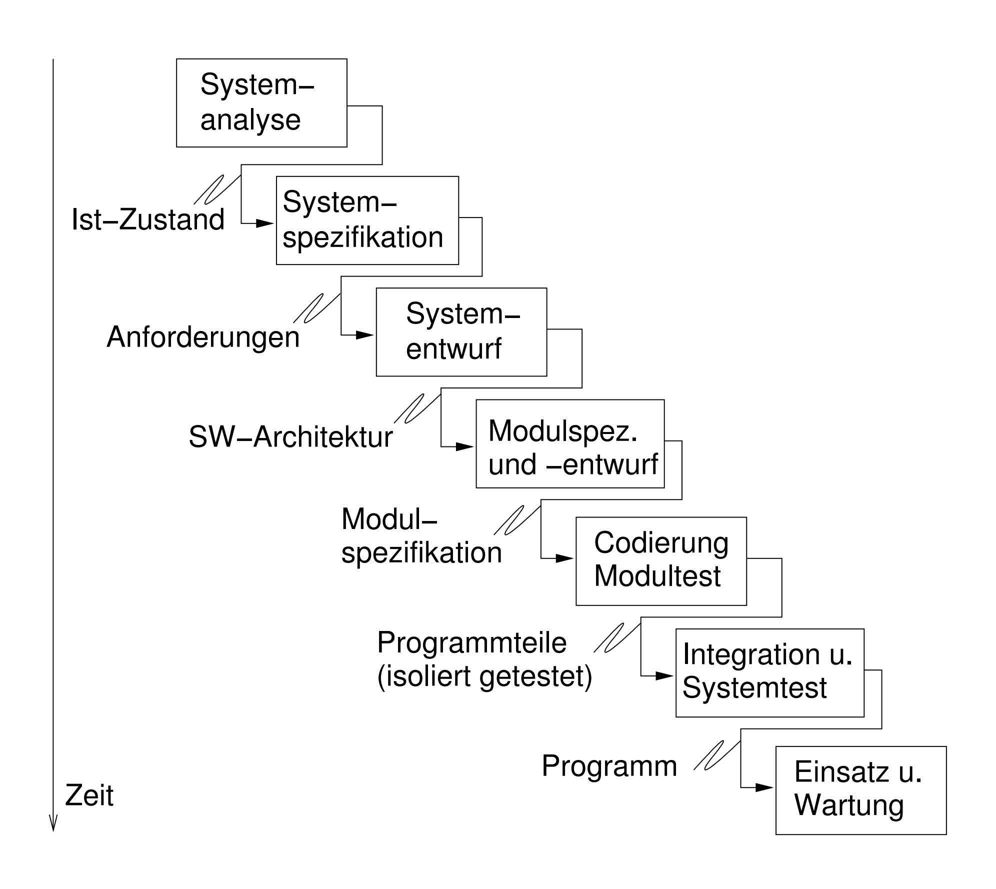
- verschiedene Software-Entwicklungsprozessmodelle kennen
- Vor- und Nachteile/Anwendbarkeit abwägen können
- die Besonderheit von Prozessen der Software-Entwicklung kennen

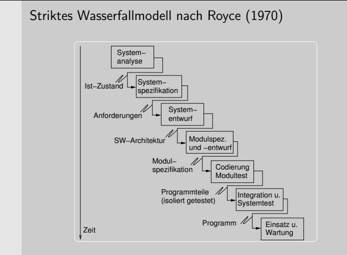
Entwicklungsprozesse

Grundlagen für guten Prozess:

- Wohldefiniertheit
  - sonst Falsch-, Nicht-, Mehrarbeit
  - sonst Information nicht verfügbar oder unverständlich
- Quantitatives Verstehen
  - objektive Grundlage für Verbesserungen
- Kontinuierliche Änderung
  - sonst Prozess schnell überholt
- Angemessenheit
  - Prozess muss dem zu lösenden Problem angemessen sein
  - d.h. effektiv und effizient sein

Striktes Wasserfallmodell nach Royce (1970)





Hier sehen wir als Beispiel das strikte Wasserfallmodell. Es enthält alle Aktivitäten in streng sequenzieller Folge. Dieses Modell eignet sich, wenn die Arbeitsschritte deutlich voneinander getrennt werden können. Alle Risiken müssen vor Projektbeginn ausgeschlossen werden können. Es ist geeignet, wenn die Mitarbeiteranzahl klein ist, da alle Mitarbeiter gleichzeitig an einem Arbeitsschritt arbeiten können.
Dieses Modell ist für die allermeisten Aufgabenstellung jedoch unrealistisch. Es setzt voraus, dass jede Aktivität auf Anhieb erfolgreich abgeschlossen werden kann. Allerdings werden z.B. die Anforderungen oft auch noch in späten Phasen des Projekts geändert bzw. erst dann überhaupt erst richtig verstanden. Dann müssen frühe Aktivitäten wiederholt werden.

Striktes Wasserfallmodell Royce (1970)

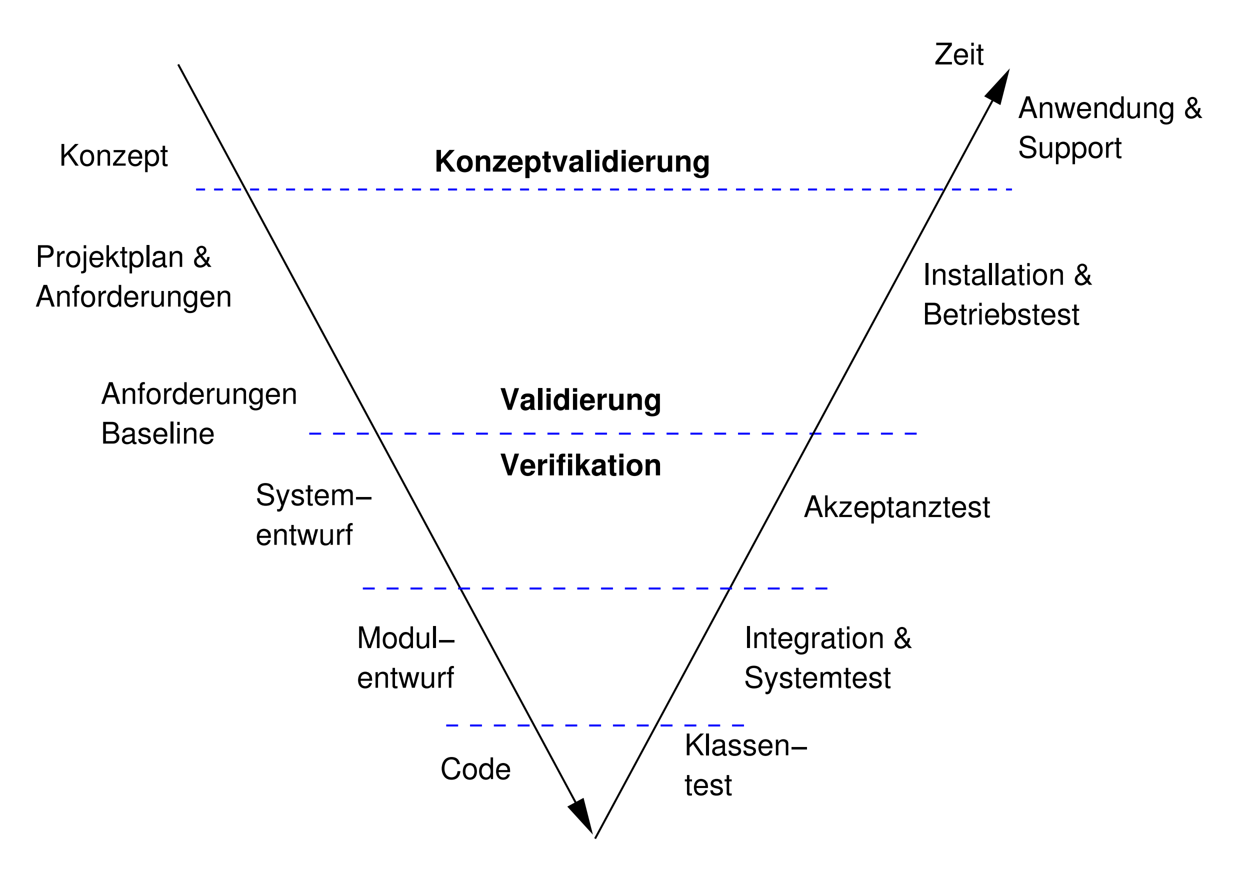
Eigenschaften dieses Modells:

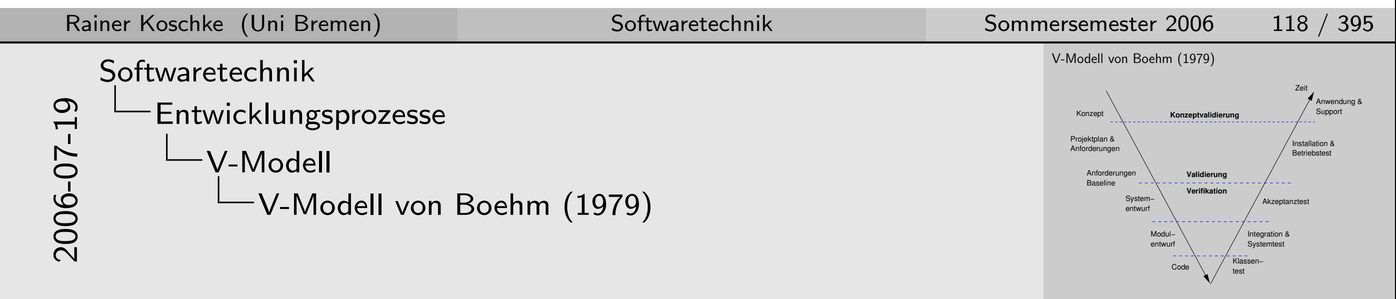
- dokumenten-getrieben: jede Aktivität erzeugt Dokument
- streng sequenzielle Aktivitäten

+ klar organisierter Ablauf
+ relativ einfache Führung
+ hohe Qualität der Dokumentation

– Entwicklung bildet langen Tunnel
– 90%-fertig-Syndrom
– Spezifikationsmängel lassen sich kaum frühzeitig erkennen
– Entwicklung beim Kunden wird ignoriert

V-Modell von Boehm (1979)





Das V-Modell ist kein eigenständiges Prozessmodell im eigentlichen Sinn. Die Produkte im V-Modell werden wie im Wasserfallmodell streng sequenziell erstellt. Es fügt dem Wasserfallmodell lediglich eine zusätzliche strukturelle Sicht hinzu, nämlich dass für jedes zu erstellende Dokument ein entsprechender Test existieren und durchgeführt werden soll.
Das V-Modell sieht sowohl Verifikationen als auch Validierungen vor. Validierung: Es wird das richtige Produkt erstellt. Verifikation: Das Produkt wird richtig erstellt.
Eine weitere Konsequenz des V-Modells für die konkrete Projektplanung ist, dass man die Test- und Validierungsaktivitäten früher als die tatsächliche Durchführung der Aktivität vorbereiten kann. Beispielsweise kann der Akzeptanztest vorbereitet werden, sobald man die Anforderungen kennt. Auf diese Weise können Aktivitäten bei einem Projektteam parallelisiert werden: Der Tester kann Testfälle für den Akzeptanztest entwickeln, auch wenn noch keine Implementierung existiert.

V-Modell von Boehm (1979)

Eigenschaften:

- entspricht Wasserfallmodell
- + betont Qualitätssicherung
- + frühe Vorbereitung von Validierung und Verifikation
  - zusätzliche Parallelisierung
  - Fehler/Mängel werden früher entdeckt
- – alle sonstigen Nachteile des Wasserfallmodells

Testgetriebene Entwicklung (Sneed 2004)

Kennzeichen testgetriebener Entwicklung:

Testfälle ...

- werden früh aus Anwendungsfällen abgeleitet
- dienen als Baseline
- treiben den Entwurf
- treiben die Kodierung

Test-Teams treiben die Entwicklung, statt von Entwicklern getrieben zu werden

Testgetriebene Entwicklung (Sneed 2004)

Anwendungs- und Testfälle

- Anwendungsfälle beschreiben Anforderungen
- sind jedoch oft nicht detailliert genug, um erwartetes Verhalten vollständig zu spezifizieren
- Testfälle können Anwendungsfälle hier komplementieren
- zu jedem Anwendungsfall sollte es mindestens einen Testfall geben
- Testfälle können zur Kommunikation zwischen Entwickler und Kunden/Anwender dienen

Testgetriebene Entwicklung (Sneed 2004)

Testfälle dienen als Baseline:

- definieren die Vorbedingungen einer Produktfunktion
- spezifizieren die Argumente einer Produktfunktion
- spezifizieren das Verhalten einer Produktfunktion (die Nachbedingungen)

Testgetriebene Entwicklung (Sneed 2004)

Testfälle treiben den Entwurf:

- Testfall ist ein Pfad durch die Software-Architektur
- Testfälle verknüpfen Anforderungsspezifikation und Architekturkomponenten
- Testfälle können zur Studie der zu erwartenden Performanz und anderer Systemattribute zur Entwurfszeit verwendet werden

Testgetriebene Entwicklung (Sneed 2004)

Testfälle treiben die Kodierung:

- vor Implementierung einer Klasse werden erst Testtreiber entwickelt
- Testtreiber enthält mindestens einen Test für jede nichttriviale Methode
- Testfall hilft, die richtige Schnittstelle zu definieren
- Testfall zwingt Entwickler, über das zu erwartende Resultat nachzudenken
- Testfälle spezifizieren die Methoden zumindest partiell

Testgetriebene Entwicklung (Sneed 2004)

Tester treiben die Entwicklung:

- Tester sind verantwortlich für die Auslieferung des Produkts
- Tester legen Kriterien für die Auslieferbarkeit fest
- Entwickler sind Lieferanten für die Tester
- Softwareentwicklung ist eine Versorgungskette; das Ende zieht, anstatt gedrückt zu werden

Bewusstseinsebenen

- bewusstes Wissen (20-30%)
  - Wissen, über das man sich im Klaren ist oder das in seiner vollen Bedeutung klar erkannt wird
- unbewusstes Wissen ($\leq$40%)
  - Wissen, das sich dem Bewusstsein im Moment nicht darbietet, aber dennoch handlungsbestimmend ist, und potenziell aufgerufen werden kann
- unterbewusstes Wissen
  - unbekannte Wünsche, die erst von außen herangetragen werden müssen, um als Anforderungen erkannt zu werden

Bewusstseinsebenen

- bewusstes Wissen (20-30%)
  - Wissen, über das man sich im Klaren ist oder das in seiner vollen Bedeutung klar erkannt wird
- unbewusstes Wissen ($\leq$40%)
  - Wissen, das sich dem Bewusstsein im Moment nicht darbietet, aber dennoch handlungsbestimmend ist, und potenziell aufgerufen werden kann
- unterbewusstes Wissen
  - unbekannte Wünsche, die erst von außen herangetragen werden müssen, um als Anforderungen erkannt zu werden

- bewusst: will mit meinem Handy telefonieren
- unbewusst: Tastatur und Display sollen auf derselben Seite meines Handys sein
- unterbewusst: ich will SMS verschicken können

Basisfaktoren

- Minimalanforderungen

→ Mangel führt zu massiver Unzufriedenheit

→ mehr als Zufriedenheit ist nicht möglich

Leistungsfaktoren

- bewusst verlangte Sonderausstattung

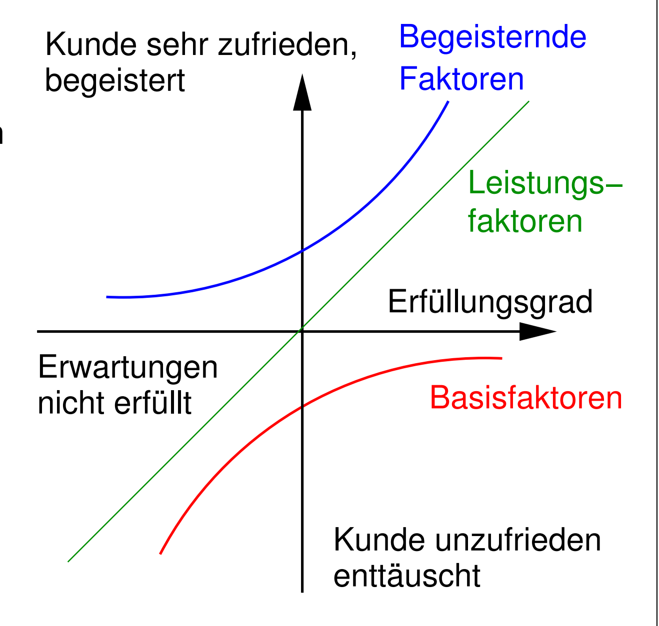
→ bei Erfüllung: Kundenzufriedenheit

→ sonst: Unzufriedenheit

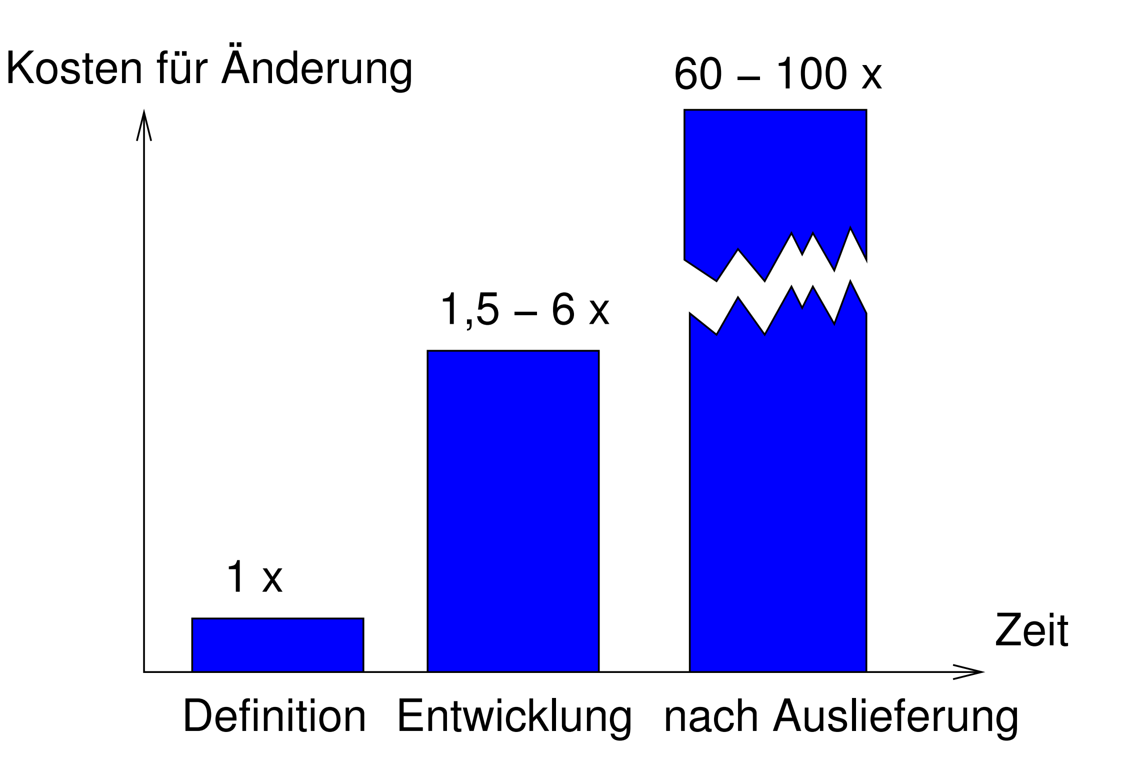
Begeisternde Faktoren

- unbewusste Wünsche, nützliche/angenehme Überraschungen

→ steigern Zufriedenheit überproportional

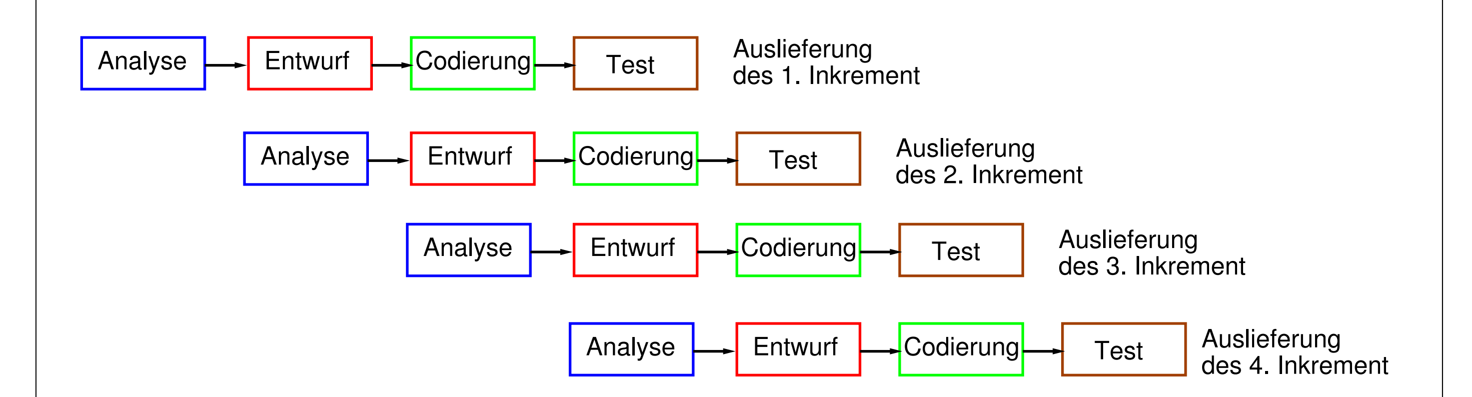


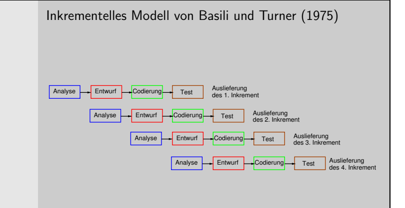
Kosten für Änderungen



Pressman (1997)

Inkrementelles Modell von Basili und Turner (1975)





Das Wasserfallmodell geht davon aus, dass alle Anforderungen zu Beginn erfasst werden können und diese während des Projekts stabil bleiben. Dies ist in der Praxis selten der Fall.
Im Laufe des Projekts gewinnen die Entwickler ein besseres Verständnis der Anwendungsdomäne und entdecken Irrtümer in ihren ursprünglichen – oft nur impliziten – Annahmen. Ähnlich wird auch der Kunde sich oft erst im Laufe des Projekts im Klaren, was er eigentlich erwarten kann und will. Auch die Rahmenbedingungen, unter denen das Projekt startete, können sich ändern (z.B. Gesetze oder Normen).
Die Anforderungen sind also selten stabil. Damit wird das Wasserfallmodell in seiner strikten Auslegung fragwürdig. Allerdings hat auch schon der Erfinder des Wasserfallmodells, Royce, empfohlen, das System zweimal zu entwickeln. Das erste Mal, um überhaupt erst Anforderungen und mögliche Lösungen auszuloten. Das zweite Mal, um ein adäquates und qualitativ hochwertiges Produkt zu erstellen.
Das inkrementelle Modell führt diesen Gedanken fort. Anstatt auf die Fertigstellung der gesamten Anforderungen zu warten, werden – sobald eine ausreichende Anzahl von Kernanforderungen beschrieben ist – für diese ein Entwurf und die Implementierung gestattet. Je mehr Anforderungen hinzukommen, desto mehr zusätzliche Inkremente werden gestartet.
Das inkrementelle Modell ist gut geeignet, wenn Kunde die Anforderungen noch nicht vollständig überblickt bzw. sich der Möglichkeiten zur Realisierung der Anforderungen nicht bewusst ist und deshalb die Anforderungen nicht formulieren kann.
Es ist mit diesem Modell möglich, Teile des Systems bereits vor Fertigstellung des gesamten Systems beim Kunden einzuführen (z.B. ein oder mehrere Subsysteme). Mit diesen Teilen kann der Kunde eine eingeschränkte Anzahl an Anforderungen realisieren. Die Zeitspanne zwischen Auftragsvergabe und Einsatz von zumindest Systemteilen wird somit geringer.
Federt die Gefahr des Wasserfallmodells ab, am Ende mit leeren Händen dazustehen. Sind wenigstens die ersten Inkremente erfolgreich entstanden, hat der Kunde zumindest ein partielles System.
Während der Entwicklung kann noch auf Änderungen reagiert werden.
Das Modell steht und fällt mit der Möglichkeit, Kernanforderungen zu identifizieren und ausreichend zu spezifizieren. Die initale Software-Architektur muss für alle Anforderungen – Kernanforderungen wie alle anderen, die im Laufe des Projekts formuliert werden – tragfähig sein; ansonsten ist ein hoher Restrukturierungaufwand notwendig. Darum sollten am Anfang alle Anforderungen bekannt sein, die sich maßgebend auf die Architektur auswirken.

Beispiel für Textverarbeitung

Iterationen:

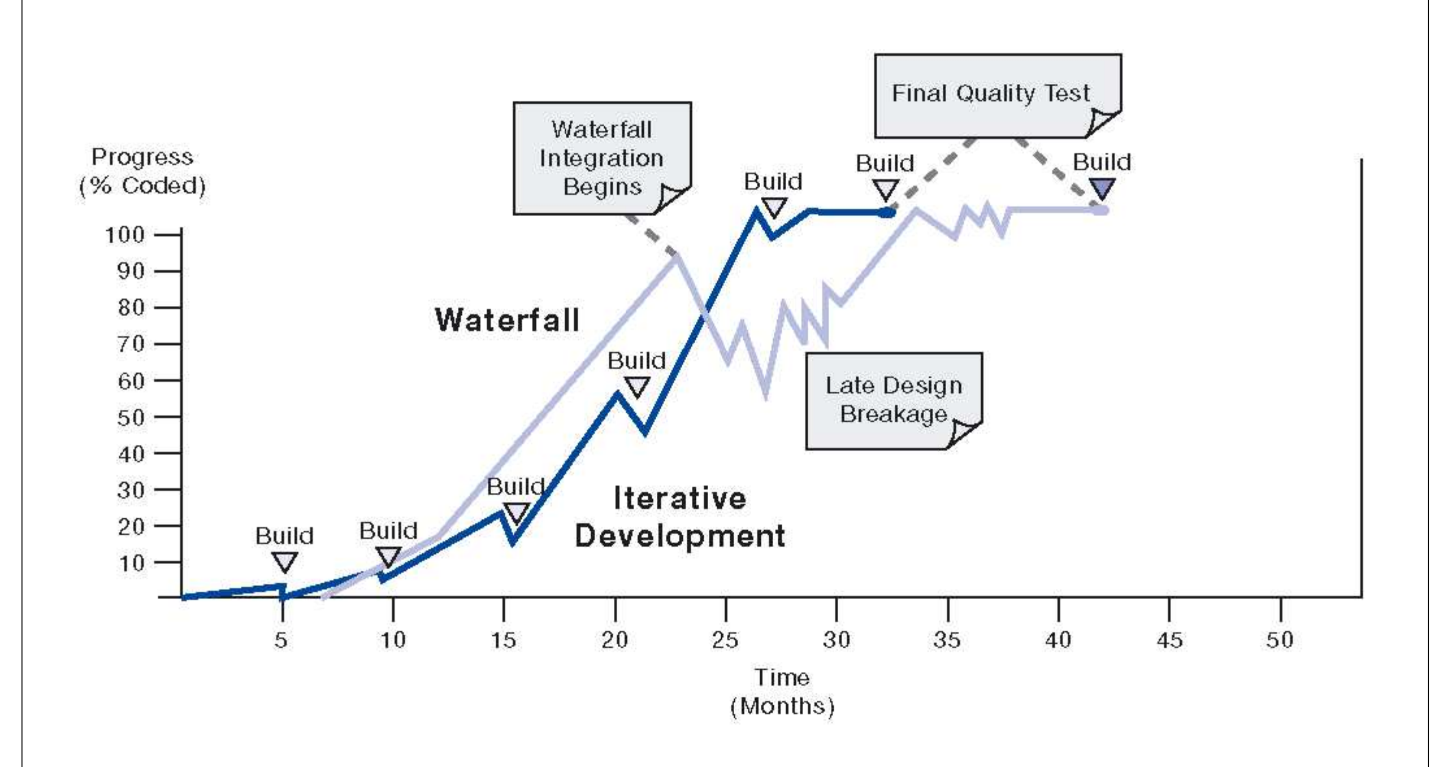
1. grundlegende Funktionalität
   - Datei-Management, Editor, Textausgabe
2. erweiterte Funktionalität
   - Style-Files, Bearbeitung mathematischer Formeln, Einbinden von Graphiken
3. zusätzliche Funktionalität
   - Rechtschreibprüfung, Grammatiküberprüfung, Überarbeitungsmodus
4. ergänzende Funktionalität
   - Tabellenkalkulation, Geschäftsgraphiken, E-Mail, Web-Browser, Scanner-Anbindung, Flipper

Inkrementelles Modell von Basili und Turner (1975)

Eigenschaften:

- Wartung wird als Erstellung einer neuen Version des bestehenden Produkts betrachtet.
- \+ Entwicklung erfolgt stufenweise
  - brauchbare Teillösungen in kurzen Abständen
- \+ Lernen durch Entwicklung und Verwendung des Systems.
- \+ Gut geeignet, wenn Kunde Anforderungen noch nicht vollständig überblickt oder formulieren kann.
  - *„I know it when I see it"*
- – Kernanforderungen und Architekturvision müssen vorhanden sein.
- – Entwicklung ist durch existierenden Code eingeschränkt.

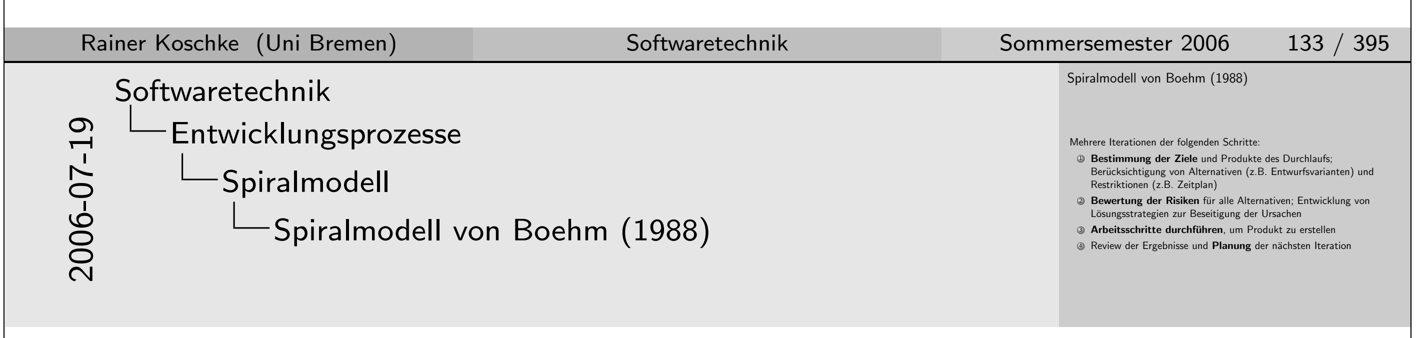
Vergleich inkrementelles Modell und Wasserfallmodell



Spiralmodell von Boehm (1988)

Mehrere Iterationen der folgenden Schritte:

1. **Bestimmung der Ziele** und Produkte des Durchlaufs; Berücksichtigung von Alternativen (z.B. Entwurfsvarianten) und Restriktionen (z.B. Zeitplan)
2. **Bewertung der Risiken** für alle Alternativen; Entwicklung von Lösungsstrategien zur Beseitigung der Ursachen
3. **Arbeitsschritte durchführen**, um Produkt zu erstellen
4. Review der Ergebnisse und **Planung** der nächsten Iteration



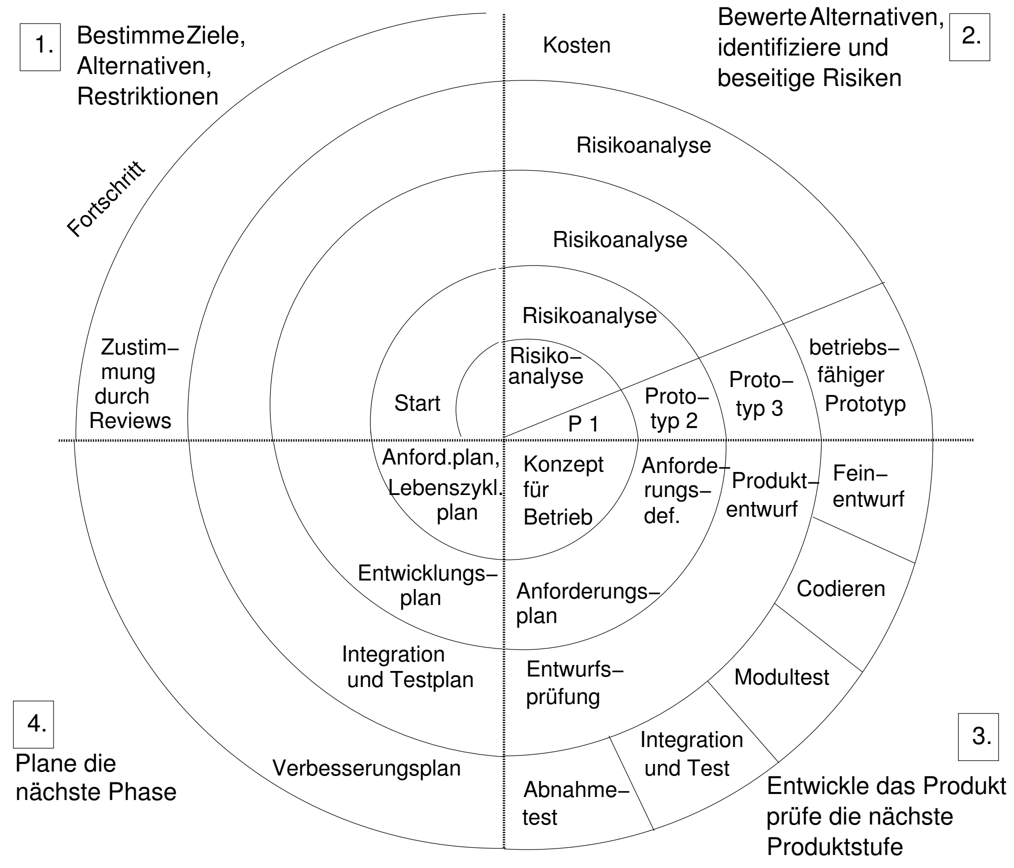
Das Spiralmodell kann für sehr große und komplexe Projekte verwendet werden, da es der Komplexität durch das risikogesteuerte Vorgehen Rechnung trägt. Die Anzahl der Durchläufe ergibt sich erst während des Projekts und wird durch die auftetenden Risiken bestimmt. Dies hat zur Folge, dass zu Beginn des Projekts ein Zeit- und Kostenplan nur schwer zu erstellen ist. Die Risikoanalyse kann nur durch erfahrene Projektleiter durchgeführt werden. Bei zu zaghaftem Vorgehen kann sich das Projekt unnätigerweise verlängern, was zu erhöhten Kosten führt. Zu schnelles Vorgehen kann Risiken vernachlässigen und zu Problemen in folgenden Durchläufen führen.

Beispiel eines Spiralmodells

(Generische) Risiken:

- Ist das Konzept schlüssig? Kann es aufgehen?
- Was sind die genauen Anforderungen?
- Wie sieht ein geeigneter Entwurf aus?

Beispiel eines Spiralmodells mit vier Durchläufen



Bewertung Spiralmodell

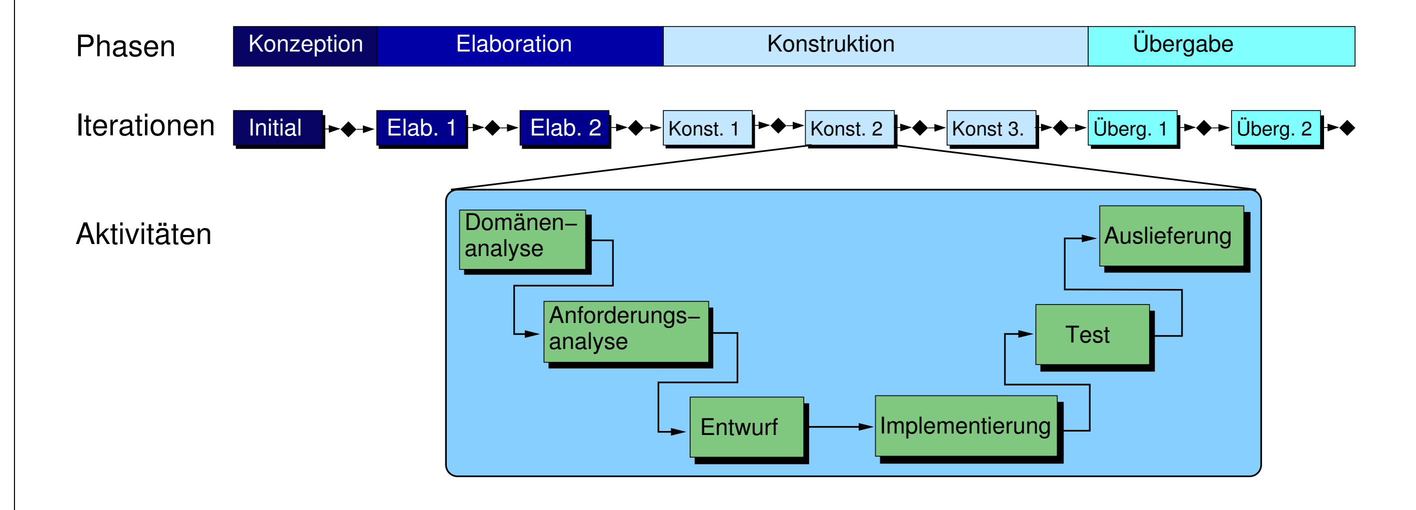
- Meta-Modell: Iterationen können beliebigen Modellen folgen

\+ bei unübersichtlichen Ausgangslagen wird die Entwicklung in einzelne Schritte zerlegt, die jeweils unter den gegebenen Bedingungen das optimale Teilziel verfolgen

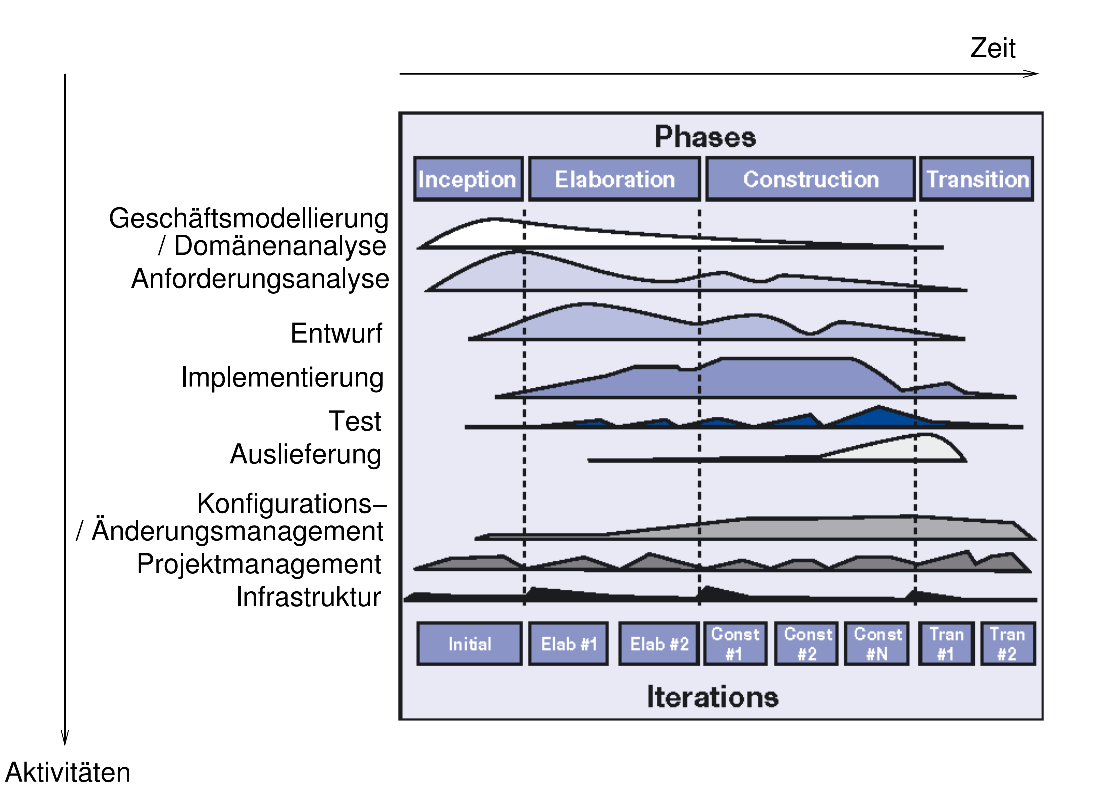
– schwierige Planung (was jedoch dem Problem inhärent ist)

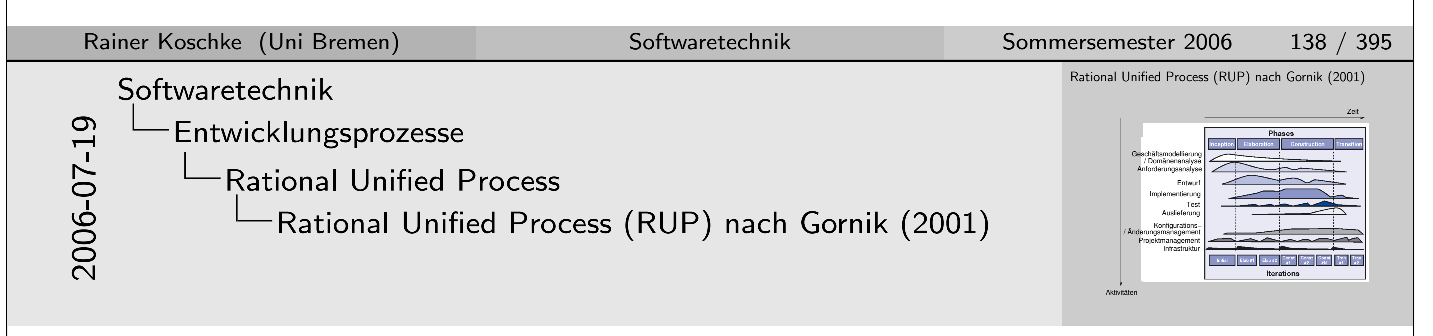
– setzt große Flexibilität in der Organisation und beim Kunden voraus

Rational Unified Process (RUP) nach Gornik (2001)



Rational Unified Process (RUP) nach Gornik (2001)



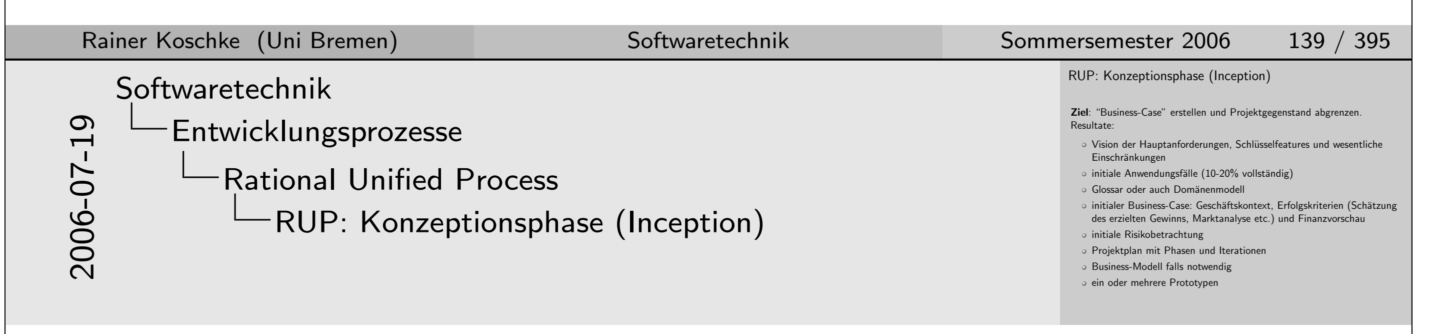


Der *Rational Unified Process* (RUP) – als Konkretisierung des *Unified Process* der Firma Rational – ist ein weiteres inkrementelles Modell.
Der Prozess insgesamt gliedert sich in vier Phasen mit einer unterschiedlichen Anzahl von Iterationen. Das Produkt jeder Phase unterscheidet sich von den Produkten anderer Phasen.
Die Konzeptionsphase (Inception) erarbeitet die Anforderungen. Die Elaborationsphase (Elaboration) erstellt einen Entwurf. Die Konstruktionsphase (Construction) erstellt das implementierte System. Die Übergabephase (Transition) führt das System beim Kunden ein.
Jede Iteration gliedert sich in die Aktivitäten Geschäftsmodellierung, Anforderungsanalyse, Entwurf, Implementierung, Test und Auslieferung. Jede Iteration wird durch ein formales Review abgeschlossen.
Die Ausprägung der einzelnen Aktivitäten ist phasenabhängig. In der Konzeptphase z.B. dient eine Implementierung lediglich einem Konzeptbeweis (Machbarkeit kritischer Anforderungen) oder einer Demonstration (ein Benutzerschnittstellenprototyp). Es genügt hierfür eine weniger aufwändige, prototypische Implementierung (der Prototyp sollte anschließend weggeworfen werden!).
Der Aufwand jeder Aktivität variiert also in den Phasen. Dies wird durch die Kurven im Schaubild veranschaulicht. Die Geschäftsmodellierung beispielsweise erzeugt in der Konzeptionsphase naturgemäß einen hohen Aufwand, hat in der Übergabephase aber keine Bedeutung mehr. Alle Flächen gemeinsam ergeben den Gesamtaufwand des Projekts. Die Summe der Flächen pro Spalte ist der Aufwand pro Iteration. Die Summe der Flächen pro Zeile ist der Aufwand pro Aktivität.
Die Hauptaktivitäten sind Geschäftsmodellierung (optional), Anforderungsanalyse, Entwurf, Implementierung, Test, Auslieferung. Die Geschäftsmodellierung dient dazu, eine gemeinsame Sprache für die unterschiedlichen Gruppen Softwareentwickler und Betriebswirte zu finden und die Software-Modelle auf die zugrunde liegenden Geschäfsmodelle zurückzuführen. Die Geschäftsprozesse werden durch so genannte Geschäfts-Anwendungsfälle dokumentiert. Sie zielen darauf ab, ein gemeinsames Verständnis, welche Geschäftsprozesse in der Organisation unterstützt werden sollen, aller Beteiligten zu erreichen. Die Geschäfts-Anwendungsfälle werden analysiert, um zu verstehen, wie die Geschäftsprozesse unterstützt werden sollen.

RUP: Konzeptionsphase (Inception)

Ziel: “Business-Case” erstellen und Projektgegenstand abgrenzen.
Resultate:

- Vision der Hauptanforderungen, Schlüsselfeatures und wesentliche Einschränkungen
- initiale Anwendungsfälle (10-20% vollständig)
- Glossar oder auch Domänenmodell
- initialer Business-Case: Geschäftskontext, Erfolgskriterien (Schätzung des erzielten Gewinns, Marktanalyse etc.) und Finanzvorschau
- initiale Risikobetrachtung
- Projektplan mit Phasen und Iterationen
- Business-Modell falls notwendig
- ein oder mehrere Prototypen

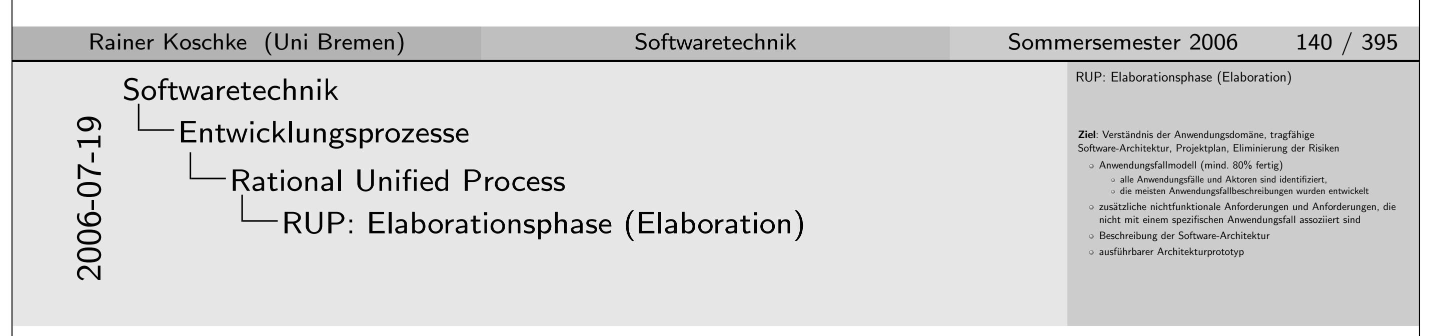


Begleitende Aktivitäten sind das Konfigurations- und Änderungsmanagement und das Projektmanagement. Ihr Aufwand ist in allen Phasen mehr oder minder gleich. Der Aufwand für das Konfigurations- und Änderungsmanagement zeigt leichte Peaks im Übergang von einer Phase zur anderen, wenn die Konfigurationen fest gezurrt werden und zum Teil nachgearbeitet werden muss.
Dem Glossar kommt eine ganz besondere Bedeutung zu. Es erklärt die Begriffe der Anwendungsdomäne. Software-Entwickler sind Spezialisten für die Entwicklung von Software, aber Laien in sehr vielen ihrer Anwendungsdomänen. Darüber hinaus verwenden auch Kunden oft die Begriffe uneinheitlich bzw. geläufige Worte mit einer ganz speziellen Bedeutung in ihrem Kontext. Mißverständnisse zwischen Kunden und Softwareentwickler sind sehr häufig und können zu teuren Fehlentwicklungen führen.
Eine Marssonde, bei deren Entwicklung europäische und amerikanische Organisationen mitwirkten, verfehlte ihr Ziel, weil den Organisationen nicht bewusst war, dass sie unterschiedliche metrische Systeme für ihre Software zugrunde legten. Die einen interpretierten einen Wert in Zentimetern, die anderen in Zoll (Inch).
Im Glossar beschreibt der Software-Entwickler, was die Begriffe des Kunden bedeuten, ebenso wie seine eigenen speziellen Begriffe. Der Kunde begutachtet das Glossar. Damit definieren beide Partein ihr Vokabular. Mißverständnisse sollen so minimiert werden.
Das Domänenmodell (oft auch konzeptuelles Modell genannt) beschreibt die Begriffe/Objekte der Anwendungsdomäne und ihre Relationen.
Eine Reihe der genannten Punkte wird sicherlich in Zusammenarbeit mit Betriebswirten ausgearbeitet. Softwareentwickler haben eine Liebe zur Technologie, übersehen jedoch leider oft die Wirtschaftlichkeit ihrer Ideen. Mit ihr steht und fällt jedoch das Projekt.
Die Erstellung von Prototypen hat das Ziel, mögliche technologische Risiken auszuschließen, dem Kunden ein konkretes Bild der möglichen Anwendung zu vermitteln (Benutzerschnittstellenprototyp), Anforderungen zu konkretisieren (“I know it when I see it”) und die Machbarkeit bestimmter Anforderungen zu demonstrieren.
Das Business-Modell erläutert, wie das System eingesetzt wird, um damit Profite zu erzielen.

RUP: Elaborationsphase (Elaboration)

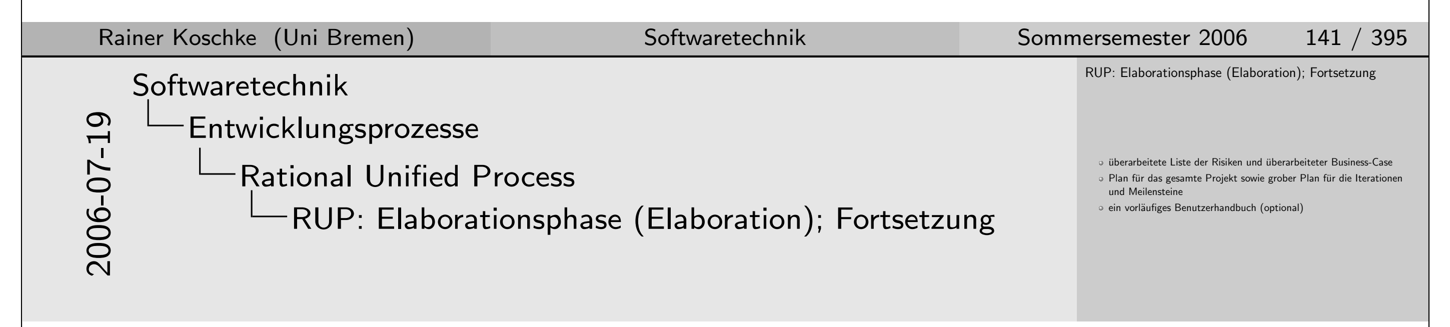
Ziel: Verständnis der Anwendungsdomäne, tragfähige Software-Architektur, Projektplan, Eliminierung der Risiken

- Anwendungsfallmodell (mind. 80% fertig)
  - alle Anwendungsfälle und Aktoren sind identifiziert,
  - die meisten Anwendungsfallbeschreibungen wurden entwickelt
- zusätzliche nichtfunktionale Anforderungen und Anforderungen, die nicht mit einem spezifischen Anwendungsfall assoziiert sind
- Beschreibung der Software-Architektur
- ausführbarer Architekturprototyp



Die Elaborationsphase ist die kritischste Phase. Hier entscheidet sich, ob das System tatsächlich gebaut wird. Der Engineering-Anteil ist weitgehend erbracht. Bis dahin halten sich die Kosten noch in Grenzen. Nun schließen sich die teure Konstruktions- und Übergabephase an.
Die Aktivitäten der Elaborationsphase stellen sicher, dass die Software-Architektur, die Anforderungen und Pläne hinreichend stabil sind (mögliche Änderungen sind antizipiert, völlig ausschließen lassen sie sich meist nicht) und Risiken sind ausreichend betrachtet, so dass Kosten und Zeitplan zuverlässig geschätzt werden können. Ab hier sollte man sich auf eine Projektdurchführung mit festem Preis einlassen können.
In der Elaborationsphase wird ein ausführbarer Architekturprototyp in ein oder mehr Iterationenj erstellt. Die Anzahl der Iterationen hängt vom Scope, der Größe, der Risiken und dem Grad des Unbekannten des Projekts ab. Zumindest die kritischen Anwendungsfälle sollten hierfür einbezogen werden, da sie typischerweise die größten technischen Risiken aufwerfen. Ein evolutionärer Prototyp (d.h. einer der schrittweise ausgebaut wird) kann durchaus verwendet werden. Man sollte jedoch auch einige explorative Wegwerfprototypen in Erwägung ziehen, um spezifische Risiken wie z.B. Entwurfs- oder Anforderungskompromisse auszuloten. Sie dienen auch als Machbarkeitsstudien und Demonstrationen.

- überarbeitete Liste der Risiken und überarbeiteter Business-Case
- Plan für das gesamte Projekt sowie grober Plan für die Iterationen und Meilensteine
- ein vorläufiges Benutzerhandbuch (optional)

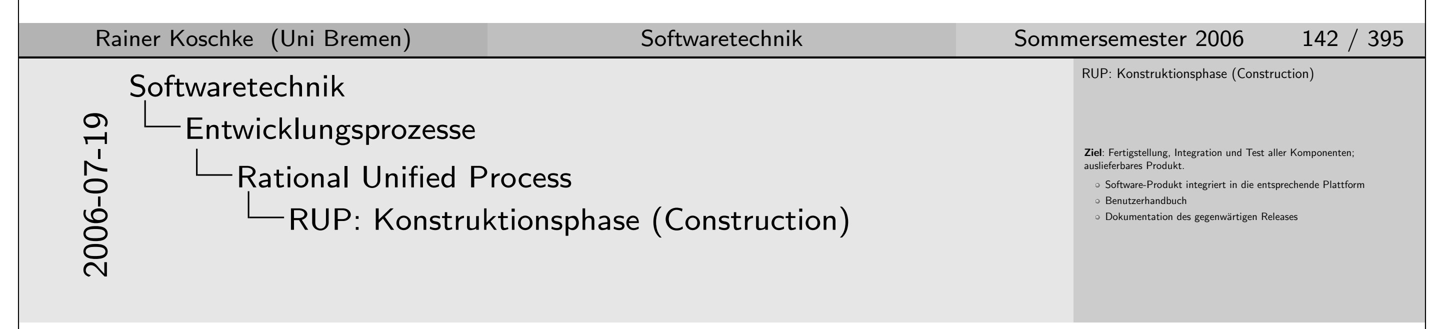


Die Erstellung des Benutzerhandbuchs kann bereits frühzeitig beginnen. Es ist konkreter als die Anforderungsspezifikation und abstrakter als die Implementierung. Somit kann es als Brücke von den Anforderungen zur Implementierung benutzt werden.
Das vorläufige Handbuch dient sowohl als Spezifikation für die Implementierung und den Test als auch für die intensivere Auseinandersetzung mit der Benutzerführung (Beispiel: Was orthogonal und einfach zu beschreiben ist, kann oft auch orthogonal und einfach implementiert werden). Überlegungen zur Benutzerführung sollten frühzeitig gemacht werden, weil Änderungen größere Restrukturierungen nach sich ziehen können.

RUP: Konstruktionsphase (Construction)

Ziel: Fertigstellung, Integration und Test aller Komponenten; auslieferbares Produkt.

- Software-Produkt integriert in die entsprechende Plattform
- Benutzerhandbuch
- Dokumentation des gegenwärtigen Releases



In der Konstruktionsphase werden alle übrigen Komponenten und Feature realisiert und in das Produkt integriert und intensiv getestet. Die Konstruktionsphase ist einem gewissen Sinne ein Herstellungsprozess, bei dem Wert auf das Management von Ressourcen und das Controlling gelegt wird, um Kosten, Zeitplan und Qualität zu optimieren. In diesem Sinne geht der Prozess nun über von der intellektuellen Entwicklung zur Entwicklung auslieferbarer Produkte.
Viele Projekte sind groß genug, um die Konstruktion zu parallelisieren. Die Parallelisierung kann die Verfügbarkeit auslieferbarer Releases zu beschleunigen. Andererseits kann sie auch die Komplexität der Ressourcenverwaltung und der Synchronisation der Arbeitsflüsse erhöhen.
Eine robuste Architektur und ein verständlicher Plan hängen stark zusammen. Deshalb ist eine kritische Qualität der Architektur die Einfachheit ihrer Konstruktion. Dies ist einer der Gründe, weshalb die ausgeglichene Entwicklung der Architektur und des Plans während der Elaborationsphase so sehr betont wird.
Das Resultat der Konstruktionsphase ist ein Produkt, das tatsächlich in die Hände des Benutzers übergehen kann.

RUP: Übergabephase (Transition)

Ziel: Produkt wird der Benutzergemeinde übergeben.
Hauptziele im Einzelnen:

- Benutzer sollten sich möglichst alleine zurecht finden.
- Beteiligte sind überzeugt, dass die Entwicklungs-Baselines vollständig und konsistent mit den Evaluationskriterien für die Vision sind.
- Erreichung der letzten Produkt-Baseline so schnell und kostengünstig wie möglich.

RUP: Übergabephase (Transition)

Typische Tätigkeiten:

- "Beta-Test", um das neue System gegen die Benutzererwartungen zu validieren
- Parallele Verwendung mit einem Lagacy-System, das durch das Produkt ersetzt werden soll
- Konversion aller notwendigen Daten (Dateien und Datenbanken)
- Schulung aller Benutzer und Administratoren
- Übergabe an Marketing, Vertrieb und Verkäufer

RUP: Übergabephase (Transition)

Typische Tätigkeiten:

- "Beta-Test", um das neue System gegen die Benutzererwartungen zu validieren
- Parallele Verwendung mit einem Lagacy-System, das durch das Produkt ersetzt werden soll
- Konversion aller notwendigen Daten (Dateien und Datenbanken)
- Schulung aller Benutzer und Administratoren
- Übergabe an Marketing, Vertrieb und Verkäufer

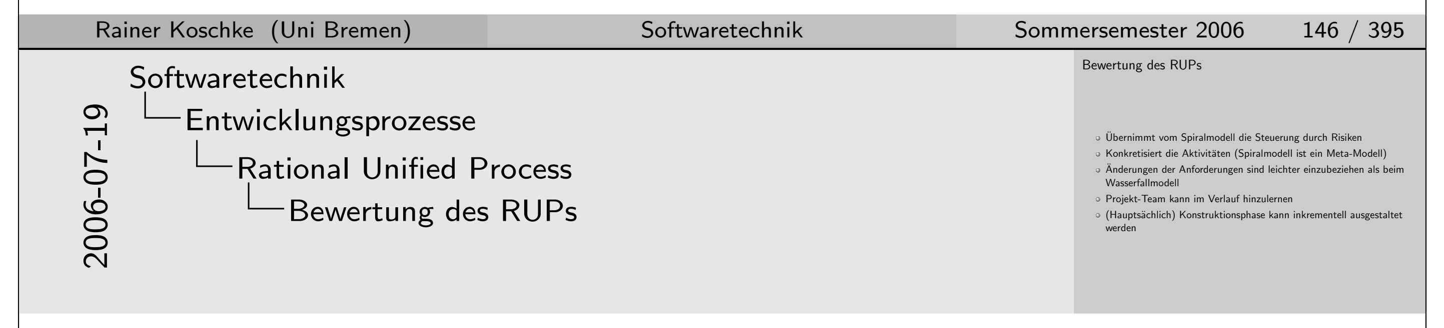
Wenn ein Produkt ausgeliefert wird, ergibt sich in der Regel schnell die Notwendigkeit neuer Releases, um Probleme zu beseitigen, Features zu realisieren, deren Implementierung verschoben werden musste, und Erweiterungen vorzunehmen.
Die Übergabephase beginnt, wenn eine Baseline reif genug ist, um beim Endbenutzer installiert werden zu können. Das erfordert typischerweise, dass zumindest eine nützliche Teilmenge des Systems mit einer aktzeptablen Qualität fertig gestellt werden konnte und dass die Benutzerdokumentation vorhanden ist.
Meist fallen mehrere Iterationen in der Übergabephase an: Beta-Releases, allgemeine Releases, Bug-Fix- und Erweiterungsreleases. Hoher Aufwand wird in die Entwickler der benutzerorientierten Dokumentation, die Schulung von Benutzern, Unterstützung der Benutzer in ihren ersten Verwendungen des Produkts und Reaktion auf Benutzer-Feedback investiert.
Man sollte sich an diesem Punkt jedoch auf den Feinschlief, die Konfiguration, Installation und Usability-Aspekte beschränken. Gänzlich neue Erweitungen sollten durch einen nachfolgenden separaten Entwicklungszyklus realisiert werden.
Die Übergabephase kann trivial sein (Software und Handbuch wird auf einen Server im Internet gelegt) oder auch sehr aufwändig und kompliziert (Ersetzung einer Flugaufsichts-Software).

Empfohlene Anzahl von Iterationen nach Kruchten (1998)

Komplexität	niedrig	normal	hoch
Konzeption	0	1	1
Elaboration	1	2	3
Konstruktion	1	2	3
Übergabe	1	1	2
Summe	3	6	9

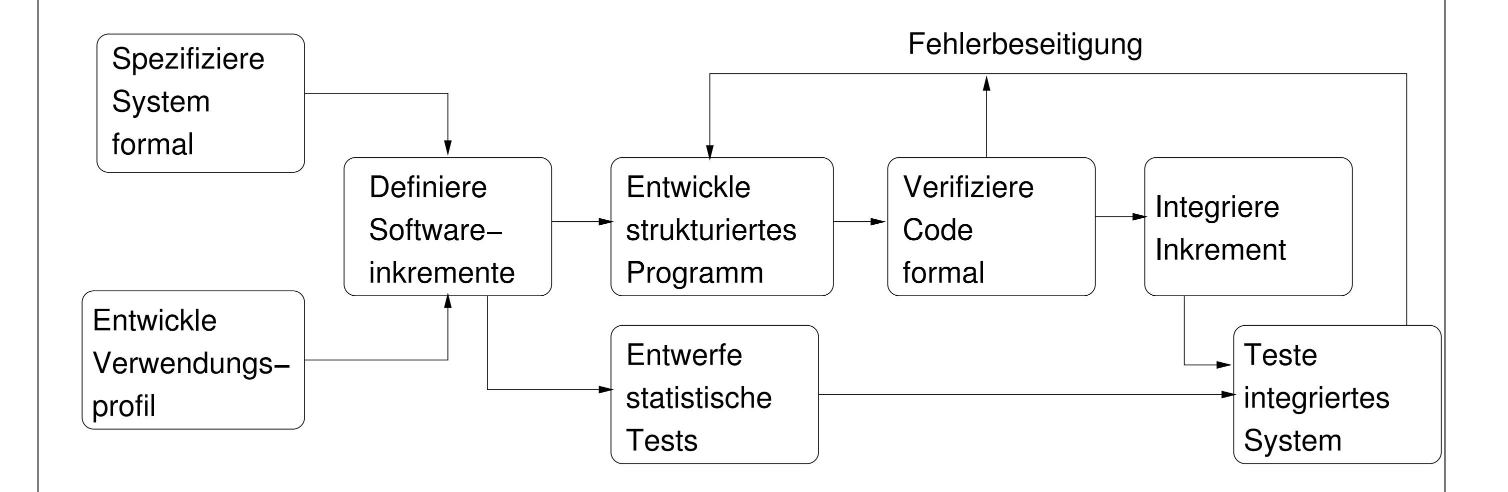
Bewertung des RUPs

- Übernimmt vom Spiralmodell die Steuerung durch Risiken
- Konkretisiert die Aktivitäten (Spiralmodell ist ein Meta-Modell)
- Änderungen der Anforderungen sind leichter einzubeziehen als beim Wasserfallmodell
- Projekt-Team kann im Verlauf hinzulernen
- (Hauptsächlich) Konstruktionsphase kann inkrementell ausgestaltet werden



Iterativ ist nicht gleich inkrementell. Bei der iterativen Entwicklung werden Entwicklungsschritte wiederholt ausgeführt. Bei der inkrementellen Entwicklung geschieht dies auch, jedoch immer um eine neues Release auf Basis eines vorherigen zu bauen. Letzteres ist im Begriff *Iteration* nicht eingeschlossen.

Cleanroom Development (Mills u. a. 1987)



Cleanroom Development

Schlüsselstrategien

- Formale Spezifikation
- Inkrementelle Entwicklung
- Strukturierte Programmierung
- Statische Verifikation
- Statistisches Testen
  - basiert auf Verwendungsprofilen (die Verwendungsweise der Software in der Praxis)
  - die häufigsten (und kritischsten) Verwendungsarten werden verstärkt getestet

Cleanroom Development

Gruppen:

- Spezifikationsteam:
  - verantwortlich für Entwicklung und Wartung der Systemspezifikation
  - erstellt kundenorientierte und formale Spezifikation
- Entwicklungsteam:
  - verantwortlich für Entwicklung und Verifikation der Software
  - Software wird nicht ausgeführt hierzu!
  - verwendet Code-Inspektion ergänzt durch Korrektheitsüberlegungen (nicht streng formal)
- Zertifizierungsteam:
  - verantwortlich für statistische Tests

Cleanroom Development

Erfahrungen Cobb und Mills (1990):

- weniger Fehler als bei traditioneller Entwicklung
- bei vergleichbaren Kosten

Extreme Programming (Beck 2000)

Extreme Programming (XP) ist eine agile Methode für

- kleinere bis größere Entwicklerteams (max. 10-15 Personen),
- Probleme mit vagen Anforderungen
- und Projekte, bei denen ein Kundenrepräsentant stets greifbar ist.

`http://www.extremeprogramming.org/`

Extreme Programming (Beck 2000)

Anerkannte Prinzipien und Praktiken werden „extrem“ umgesetzt:

- Code-Reviews → permanente Reviews durch Pair-Programming
- Testen → ständiges Testen: Unit-Tests sowie Akzeptanztests durch den Kunden/Benutzer
- klare Struktur → jeder verbessert sie kontinuierlich durch Refactoring
- Einfachheit → stets die einfachste Struktur wählen, die die aktuellen Anforderungen erfüllt
- Integration → permanente Integration auch mehrmals am Tag
- Validierung:
  - Kunde/Benutzer ist stets involviert bei der Planung neuer Iterationen und verantwortlich für Akzeptanztest
  - kurze Iterationen → Dauer in Minuten und Stunden, nicht Wochen, Tage, Jahre

aber auch Auslassung anerkannter Prinzipien:

- Dokumentation: mündliche Überlieferung, Tests und Quellcode
- Planung: sehr begrenzter Horizont

Weitere XP-Charakteristika

- Kunde vor Ort
- eine Metapher statt einer Architekturbeschreibung
- 40-Stundenwoche
- Code ist kollektives Eigentum
- Kodierungsstandards

Agile versus weit voraus planende Prozessmodelle (Boehm und Turner 2003)

Risiken agiler Methode:

- Skalierbarkeit, Kritikalität, Einfachheit des Entwurfs, Personalfluktuation, Personalfähigkeiten

Risiken weit voraus planender Prozessmodelle:

- Stabilität der Anforderungen, steter Wandel, Notwendigkeit schneller Resultate, Personalfähigkeiten

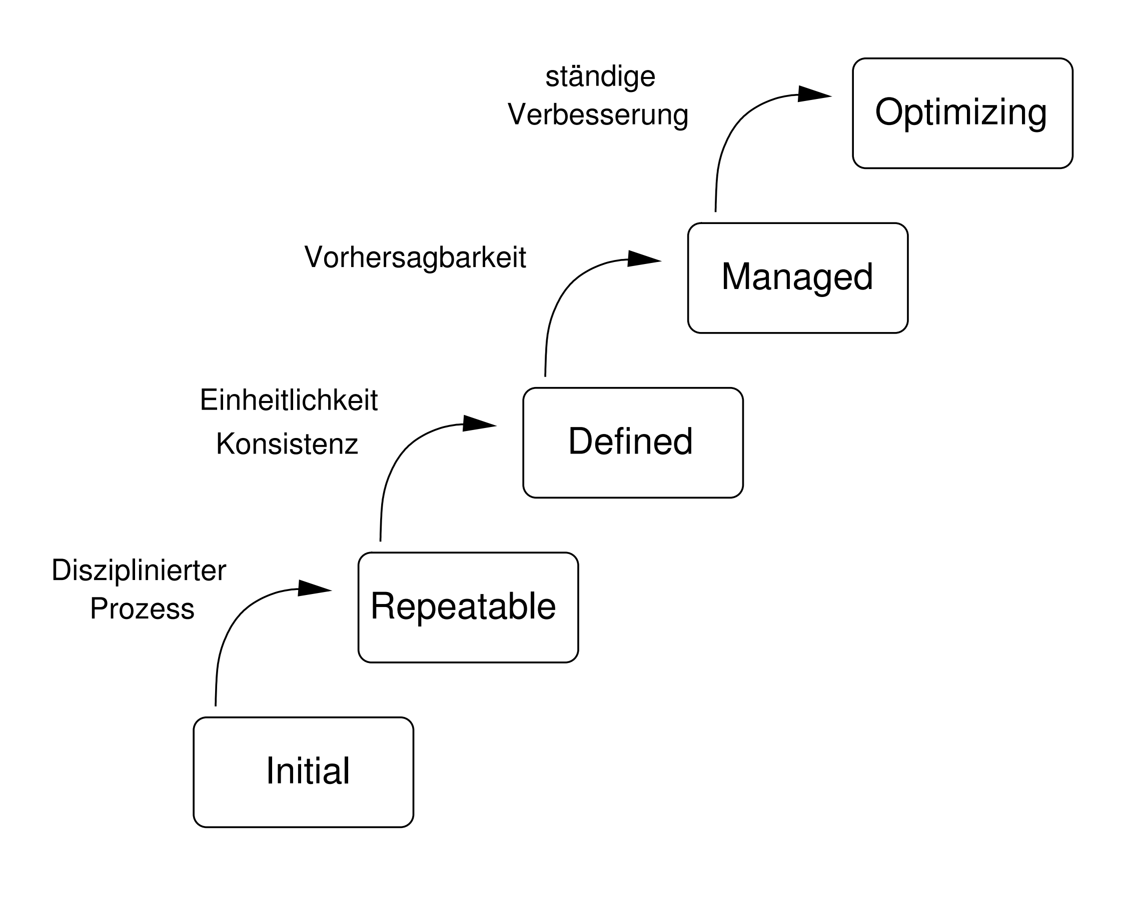
Generelle Risiken:

- Unsicherheiten bei der Technologie, unterschiedliche Interessengruppen, komplexe Systeme

Capability Maturity Model

- Entwickelt vom SEI 1985-91 für DoD
- Beschreibt Stufen der Prozessreife
- Maßstab/Leitfaden für Verbesserungen
- Idee: besserer Prozess → besseres Produkt
- 5 Stufen (CMM Level 1-5)
- Definiert Schlüsselbereiche (Key Process Areas)
- Steigende Transparenz des Prozesses

Capability Maturity Model



CMM Level 1 – Initial

- Prozess meist völlig instabil
- Typisch: in Krisen nur noch Code & Fix
- Qualität und Fertigstellung unvorhersagbar
- Erfolge nur durch gute Leute und großen Einsatz

CMM Level 2 – Repeatable

- Projekterfolge nachvollziehbar und wiederholbar
- Projektplanung und -management basiert auf Erfahrung
- Key Process Areas:
  - Anforderungsverwaltung (u.a. Reviews)
  - Projektplanung (Zeitplanung, Risikomgmt., Prozess)
  - Projektverfolgung und -überblick (Transparenz)
  - Unterauftragsverwaltung
  - Qualitätssicherung (QS-Plan)
  - Konfigurationsverwaltung (Konsistenz, Änderungsverfolgung)

CMM Level 3 – Defined

- Organisationsweiter Prozess, wird für jedes Projekt angepasst
- Zuständig: Software Engineering Process Group
- Kosten, Zeitplan und Funktionalität im Griff
- Key Process Areas:
  - Organisationsweiter Prozess
  - Prozessdefinition
  - Ausbildungsprogramm
  - Integriertes Softwaremanagement (Anpassung auf konkretes Projekt)
  - Software-Engineering-Techniken, Tool-Unterstützung
  - Koordination zwischen Gruppen
  - Peer Reviews

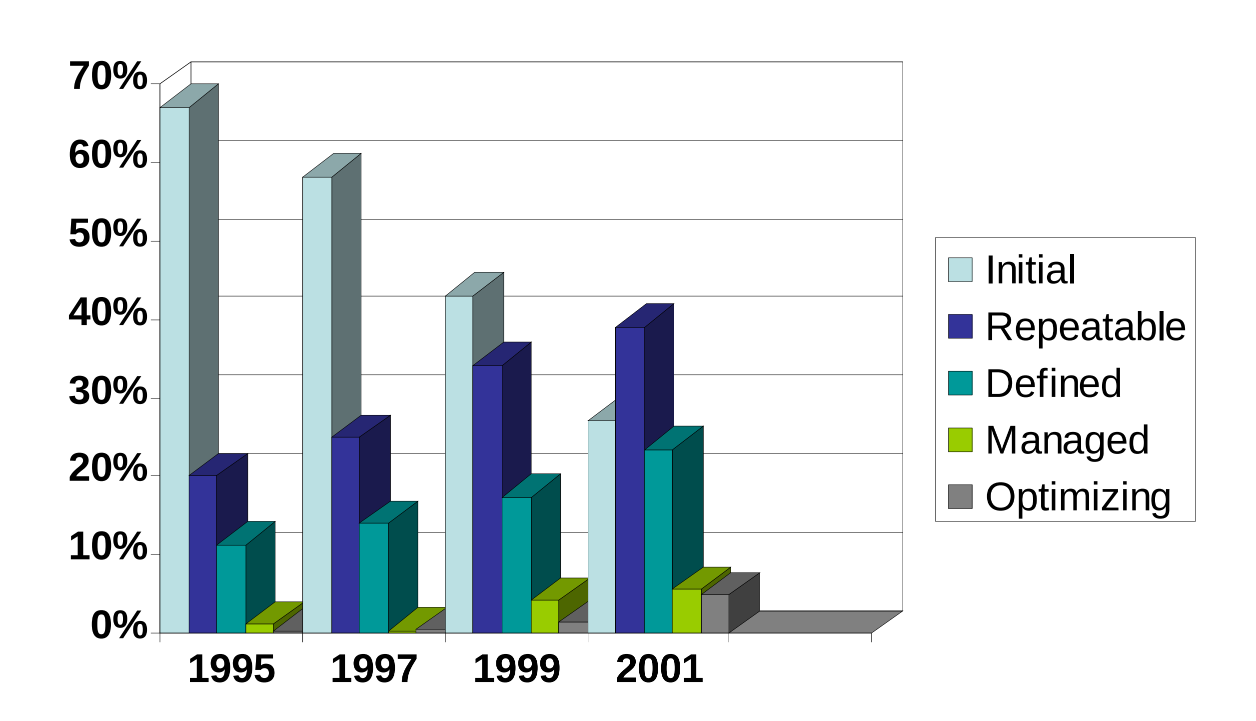
CMM Level 4 – Managed

- Einbeziehung quantitativer Aspekte
- Ziele setzen und überwachen
- Prozessmessdaten werden aufgenommen, archiviert, analysiert
- Vorhersagbarkeit steigt
- Key Process Areas:
  - Quantitative Prozesssteuerung
    $\rightarrow$ Leistung des Prozesses überwachen
  - Software-Qualitätsmanagement
    $\rightarrow$ Messbare Ziele für Prozessqualität

CMM Level 5 – Optimizing

- Änderungsmanagement für Technologie und Prozesse
- Feedback von Projekten zum Prozess → ständige Verbesserung
- Key Process Areas:
  - Defektvermeidung
    → Analyse von Fehler-/Problemursachen
    → Vermeiden erneuten Auftretens
  - Verwaltung von Technologieänderung
    → neue Technologien bewerten, evtl. einführen
  - Verwaltung von Prozessänderung
    → kontinuierliche Verbesserung

Capability Maturity Model

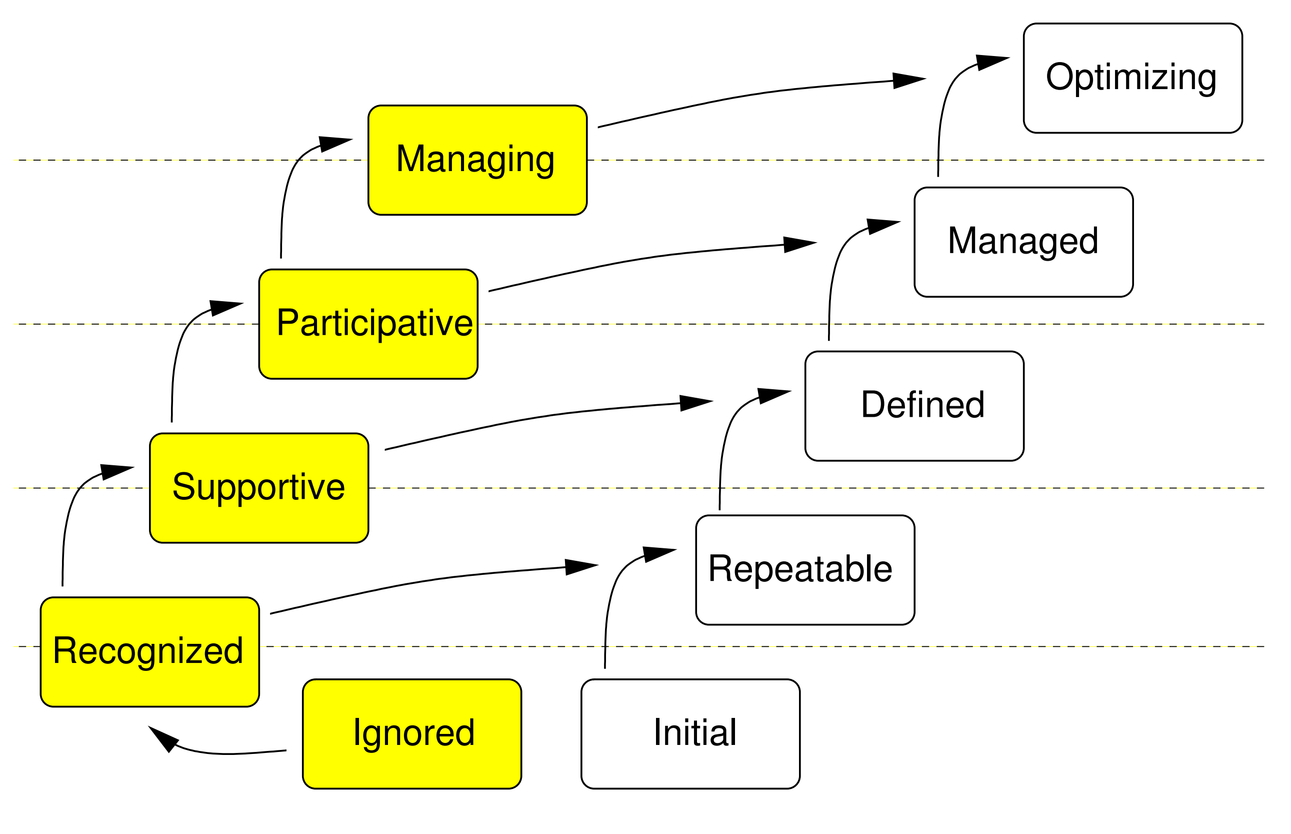


SEI Assessments (Quelle: SEI)

Probleme mit CMM

- Management: „zu teuer“
- Wenig verbreitet (ca. 10-15%)
- Nur langsame Verbesserung (ca. 2 Jahre/Level)
- Neue Technologien nicht berücksichtigt

Capability Maturity Model – Management



Quelle: Parker (1995)

Capability Maturity Model Integration (CMMI)

- Viele Maturity-Model-Varianten

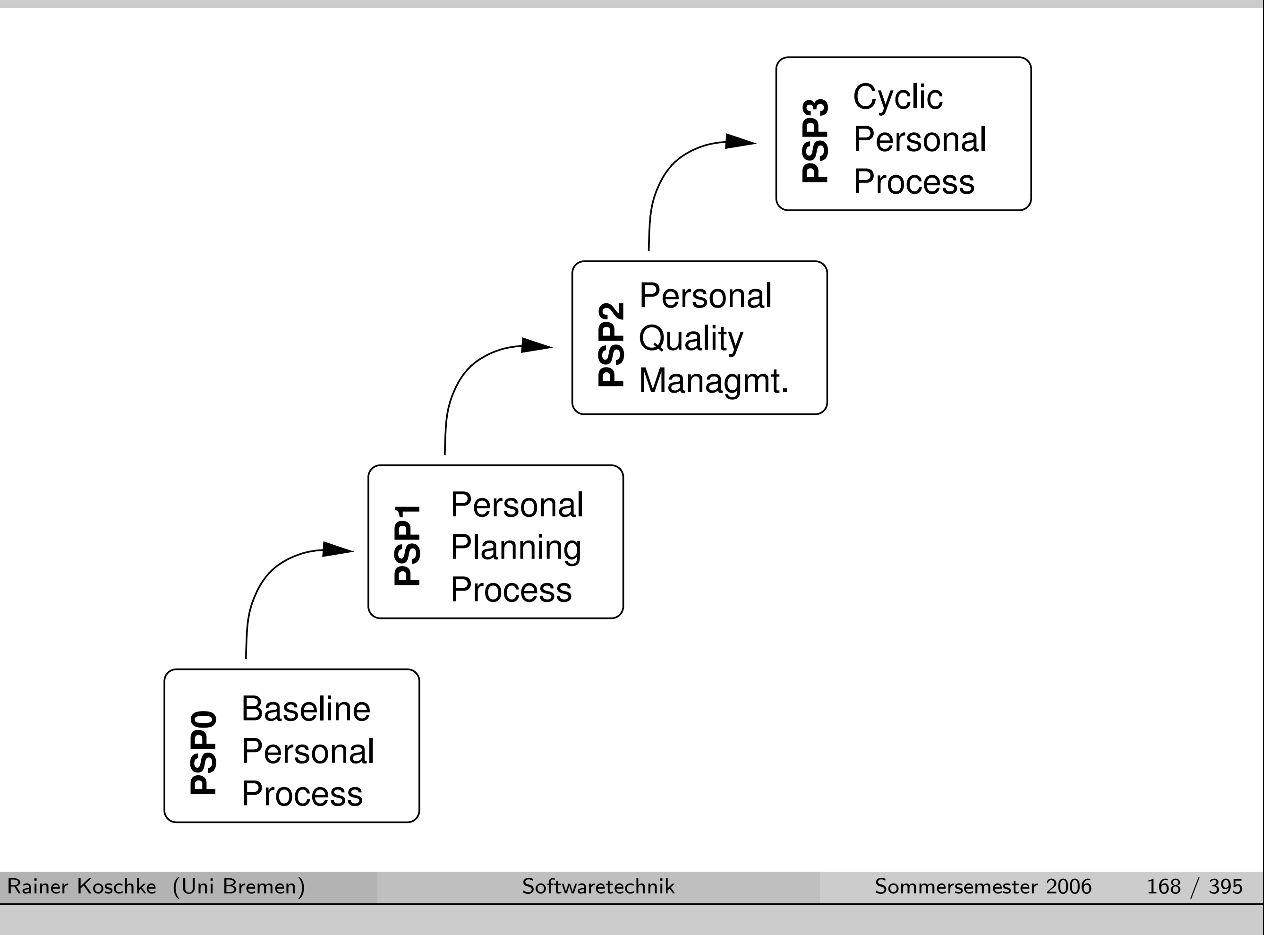
→ CMMI: „Integration“ (SEI 2000):

- Angepasst an iterative Entwicklung
- Generische Ziele hinzugefügt
- Zusätzliche KPAs:
  - Level 2: Measurement and Analysis
  - Level 3: Software Product Engineering (verfeinert); Risk Management; Decision Analysis and Resolution
  - Level 4, 5: nur Restrukturierungen

Persönlicher Softwareprozess (PSP)

- Watts S. Humphrey (1995) (SEI)
- Anwendung von CMM auf einen Entwickler
- Schwerpunkte: Planung, Qualität
- Vorteile:
  - Schneller umsetzbar
  - Konkrete Techniken angebbar
  - Verbesserungen sofort wahrnehmbar
  - → höhere Motivation

PSP – Evolution



PSP0 – Baseline Personal Process

PSP0: Bisherige Vorgehensweise plus

- Messung
  - Zeit pro Phase
  - gefundene/gemachte Fehler pro Phase
  - Zeit für Fehlerbehebung
- Formulare: Projektplan, Zeiten, Fehler

PSP0.1: plus

- Codierrichtlinien
- Messung der LOC (Veränderungen)
- Formular: Prozessverbesserung

PSP1 – Personal Planning Process

PSP1: PSP0.1 plus

- Größenschätzung mit PROBE (PROxy-Based Estimating)
  - Schätzung basiert auf Objekten (Entwurf)
  - Unterscheidet Objekte nach Typ, Größe, #Methoden
  - Daten sammeln, Regressionsanalyse
- Formular: Testbericht

PSP1.1: PSP1 plus

- Aufgabenplanung
- Zeitschätzung, -planung
- Projektverfolgung (Earned Value Tracking)

PSP2 – Personal Quality Management

PSP2: PSP1.1 plus

- Code Reviews (Checklisten)
- Design Reviews (Checklisten)

PSP2.1: PSP2 plus

- Design Templates:
  - Operational Scenario ($\approx$ Anwendungsfall)
  - Functional Specification (formale Spezifikation)
  - State Specification ($\approx$ Zustandsdiagramm)
  - Logic Specification (Pseudocode)

→ Vermeidung von Designfehlern

→ Beurteilung der Qualität

- Cost-of-Quality (Behebung, Bewertung, Vermeidung)

PSP3 – Cyclic Personal Process

PSP3:

- Anwendung auf große Projekte
- Nach High-Level Design: Aufteilung in Module
- Anwendung von PSP2.1 auf jedes Modul
- Formular: Issue tracking log

Wiederholungs- und Vertiefungsfragen

- Erläutern Sie die Ideen sowie Vor- und Nachteile der Entwicklungsprozesse...
  - Wasserfall
  - V-Modell
  - testgetriebene Entwicklung
  - inkrementelle Entwicklung
  - Spiralmodell
  - Rational Unified Process
  - Cleanroom Development
  - Extreme Programming
- Gegeben ist das folgende Szenario: [...]. Welches Vorgehensmodell empfehlen Sie?
- Unter welchen Umständen würden Sie eher agiles Vorgehen als voraus planendes empfehlen?
- Stellen Sie das Capability Maturity Model dar. Wozu dient es?
- Wie können Prozesse verbessert werden?
- Was ist der persönliche Software-Entwicklungsprozess? Wozu dient er?

Weiterführende Literatur

Bibliographie zu Entwicklungsprozessen:

Arbeitskreis der Fachgruppe 5.11
"Begriffe und Konzepte der Vorgehensmodellierung"
http://www.vorgehensmodelle.de/giak/arbeitskreise/vorgehensmodelle/publallg.html

Rational Unified Process: Kruchten (1998)

Komponentenbasierte Entwicklung

Lernziele

- Benutzung von Komponenten
  - Komponentenbasierte Entwicklung
  - Komponentenmodelle
  - Schnittstellen
  - Komponenten
  - Zusammenbau
- Erschaffung von Komponenten
  - Herstellung
  - Implementierungsaspekte

Komponente

Definition

Software components: are executable units of independent production, acquisition, and deployment that interact to form a functioning system (Szyperski u. a. 2002).

- "to compose a system"
- dazu da, um zusammengesetzt zu werden
- N.B.: Komponente $\neq$ Klasse $\neq$ Verteilung

Komponentenbasierte Entwicklung

- Trennung von Schnittstellen (liefern, benutzen) und Implementierung
- Standards für Integration
  - einheitliche Schnittstellenbeschreibung
  - unabhängig von der Programmiersprache
- Infrastruktur (Middleware)
- Entwicklungsprozess

→ technische und nicht-technische Aspekte

Motivation I

Wiederverwendung als Schlüssel:

- ökonomisch
- als Lösung der Softwarekrise
- OO konnte die Erwartungen an Wiederverwendbarkeit und Vermarktung nicht erfüllen

→ ohne Wiederverwendung: nur lineares Wachstum möglich

→ mit Wiederverwendung: superlineares Wachstum möglich (so die Hoffnung)

Motivation II

Ebenen der Wiederverwendung

- konkrete Lösungsteile: Bibliotheken
- Verträge: Schnittstellen
- Vertragsanbieter: Komponenten
- einzelne Interaktionsteile: Meldungen und Protokolle
- Architekturen für Interaktion: Muster
- Architekturen für Teilsystem: Frameworks
- Gesamtsystem: Systemarchitekturen

Maßgefertigt vs. Standardsoftware

Maßanfertigung:

- optimale Anpassung an Kunde → Wettbewerbsvorteil
- keine Änderung der Kundenprozesse notwendig
- unter lokaler Kontrolle

Standardsoftware:

- billiger
- schneller einsetzbar
- geringeres Risiko des Scheiterns
- Wartung und Evolutionsanpassung durch den Hersteller
- leichtere Zusammenarbeit mit anderen Systemen

Vorteile von beiden Ansätzen: Maßanfertigung aus Standardkomponenten

Integration

- Problem
  - Komponenten kommen aus verschiedenen Quellen
  - Einbau der Komponenten von Dritten
  - Fehlereingrenzung schwierig
  - keine klassischen Integrationstests, da späte Integration
- System nur so stark wie schwächste Komponente
- eine Lösung: defensives Programmieren

Vor- und Nachteile I

Vorteile:

- verbesserte Qualität
- Wiederverwendung verringert die Dauer bis zur Auslieferung
- Modularisierung (nur lokale Änderungen, Abhängigkeiten explizit, ...)
- Austauschbarkeit durch Abstraktion (Schnittstellen)
- Markt: innovative Produkte, niedriger Preis,...

Vor- und Nachteile II

Nachteile:

- höhere Kosten durch:
  - komplexere Technik
  - durch Outsourcing höhere Kosten durch Riskoabstützung
  - mehr Aufwand, um vergleichbare Systemstabilität zu erreichen
- offene Probleme:
  - Vertrauenswürdigkeit der Implementierung
  - Zertifizierung der Implementierung
  - gewünschte Anforderungen vs. verfügbare Komponenten

Prozess

Anpassung des Entwicklungsprozesses:

1. Anforderungen erfassen
2. Identifizierung von Komponentenkandidaten (suchen, auswählen, überprüfen)
3. Anpassen der Anforderungen an gefundene Kandidaten
4. Design der Architektur
5. Überprüfung der Komponentenkandidaten und event. neue Suche
6. Erstellen der nichtabgedeckten Funktionalität
7. Zusammenstellen des Systems aus Komponenten mit Verbindungscode (Glue-Code)

Komponentenmodelle

- Sicherstellen der Interoperabilität
- Standards für

Schnittstellen:

- Schnittstellenbeschreibung
- spezielle Schnittstellen
- Interaktion zwischen Komponenten

Informationen zur Verwendung:

- Namensregeln
- Individualisieren
- Zugriff auf Metadaten

Einsatz:

- Verpackung
- Dokumentation
- Evolutionsunterstützung

Beispiele für Komponentenmodelle

im weiteren Sinn:

- Anwendungen in einem Betriebssystem
- Plugins
- Verbunddokumente (Office Dokumente mit OLE, HTML)

im engeren Sinn: CORBA, COM, JavaBeans

Eigenschaften erfolgreicher Komponentenmodelle

- Infrastruktur mit guter Basisfunktionalität
- Komponenten werden von unterschiedlichen Herstellern angeboten und von Kunden eingesetzt
- Komponenten von verschiedenen Anbietern arbeiten in einer Installation zusammen
- Komponenten haben eine Bedeutung für den Klienten

Allgemeine Dienste

- meist durch Komponentenmodelle als Schnittstelle festgelegt
- Anbieter der Komponentenmodelle bietet auch diese Komponenten an
- besonders wichtig für verteilte Systeme
- Beispiele:
  - Verzeichnisdienst
  - Persistenz
  - Nachrichtendienst
  - Transaktionsmanagement
  - Sicherheit

Schnittstellen

- ermöglichen:
  - Zusammenarbeit zwischen fremden Komponenten
  - Austauschbarkeit (Anbieter und Benutzer)
  - Identifizierung der Abhängigkeiten
- interner Zustand nichtöffentlich (alle Zugriffe über Schnittstelle)
- Qualität von höchster Bedeutung
- eine Komponente kann Serviceprovider für mehr als eine Schnittstelle sein (provides)
- eine Komponente kann den Service anderer Schnittstellen benötigen (requires)
- späte Integration → späte Bindung → Indirektion
- direkte (prozedural) und indirekte (Objekt-) Schnittstellen
- beschrieben durch Meta-Information zur Laufzeit, Interface Description Language (IDL) oder „direkt“ (Java)

Beispiel (CORBA-IDL)

```
module Bank {
  typedef long pin_t;
  enum KontoFehlerTyp {
    UngenuegendeKontodeckung
  };

  exception KontoException {
    KontoFehlerTyp typ;
    string beschreibung;
  };

  interface Konto {
    readonly attribute string name;
    readonly attribute long kontoStand;

    boolean isValidPin(in pin_t pin);
    void abheben(in long betrag) raises(KontoException);
    void einzahlen(in long betrag);
  };
};
```

Vertrag

- Schnittstelle als Vertrag zwischen Anbieter und Benutzer des Services
- Anbieter:
  - über Funktionalität: z.B. als Vor- und Nachbedingungen
  - über nicht-funktionale Anforderungen (Service-Level, Ressourcen); z.B. Standard Template Library für C++
  - Darstellung:
    - informal als Text
    - formaler z.B. durch temporale Logik (um Terminierung zuzusichern) oder mit OCL
- Kunde:
  - Vermeidung von speziellen Eigenschaften einer bestimmten Implementierung (d.h. nur Vertrag benutzen)

Versionen

- Problem: sowohl Schnittstelle als auch Implementierung ändern sich $\rightarrow$ unterschiedliche Versionen nicht vermeidbar
- Ziel: Entscheidung, ob kombatibel oder nicht
  - zusätzlich noch Unterstützung für einen Bereich von Versionen (sliding window)
- Lösungen (Lösungen?):
  - unveränderliche Schnittstellen
  - Schnittstellen dürfen sich ändern, aber nur nach Regeln (z.B. Parametertyp darf verallgemeinert werden)
  - Ignorieren des Problems:
    - abwälzen auf tiefere Schicht
    - immer neu kompilieren

Beschaffung

- Annahme: großer Markt von Komponenten
- Suche: Komponenten und funktionale Anforderungen werden klassifiziert
- Qualitätskriterien für Auswahl:
  - funktionale und nicht-funktionale Anforderungen
  - Schnittstellenbeschreibung
  - Abhängigkeiten und Kompatibilität
  - Vertrauenswürdigkeit, Überlebenschance des Anbieters, Garantien und Wartungszusicherung
  - Preis und Zahlungsart
  - ...

Entstehung von Komponenten

- meist aus bestehender Software
- Anpassung notwendig, um Wiederverwendbarkeit zu erreichen
- ist Investition ökonomisch sinnvoll?
  - Größe des Einsatzgebiets
  - bislang angebotene Funktionalität
  - potentieller Grad der Wiederverwendung

Balance zwischen

großen Komponenten

- bieten mehr Service an
- weniger Abhängigkeiten
- schneller, da keine cross-context Aufrufe

kleinen Komponenten

- verständlicher
- billiger für den Benutzer
- mehr Freiheit für den Benutzer
- mehr Benutzer

Implementierung

- defensives Programmieren, da keine Integrationstests
- wiederverwendbar machen:
  - entferne anwendungsspezifische Methoden
  - generalisiere Namen
  - füge Methoden hinzu, um Funktionalität zu vervollständigen
  - führe konsistente Ausnahmebehandlung ein
  - füge Möglichkeiten hinzu, die Komponenten an verschiedene Benutzer anzupassen
  - binde benötigte Komponenten ein, um die Unabhängigkeit zu erhöhen

Typen

- Typ als vereinfachter Vertrag mit besserer Überprüfbarkeit
- Überprüfung des Vertrages (Übersetzungszeittest besser als Ladezeittest besser als Laufzeittest besser als kein Test)
- Subtypen
  - Austauschbarkeit (Liskovsches Substitutionsprinzip); deklarative Subtypen (aka Vererbung) vs. strukturelle Subtypen
  - Eingabeparameter gleicher Typ oder Supertyp (Kontravarianz)
  - Ausgabeparameter gleicher Typ oder Subtyp (Kovarianz)
- Implementierungen können den Vertrag ändern: nur Vorbedingungen abschwächen oder Nachbedingungen verschärfen

Vererbung

- Anpassen einer Komponente:
  - durch Parametrisieren und Verbinden mit anderen Komponenten
  - durch Ableiten
- Arten:
  - Implementierungsvererbung
  - Schnittstellenvererbung
- Zusage der Austauschbarkeit
- Mehrfachvererbung:
  - Schnittstellenvererbung: kein Problem
  - Implementierungsvererbung: problematisch

Zerbrechliche Basisklassen

- Basisklasse wird geändert, sind erbende Klassen immer noch funktionsfähig?
- unvorhergesehene Aufrufgraphen → unerwarteter Wiedereintritt
- notwendig ist Schnittstellenvertrag zwischen Klasse und erbenden Klassen
- Lösungen:
  - Schutz (public, protected, private, finale, override)
  - disjunkte Gruppen von Methoden und Variablen
  - alle Instanzvariablen sind privat und die erbende Klasse fügt keine hinzu
  - Ableiten der Klasse verbieten

Aufrufe zwischen Komponenten in verteilten Systemen

- Idee des lokalen Aufrufes bleibt erhalten
- Generierung von Stubs
- Aktionen des Aufrufers bei Aufruf:
  1. Kontrolle geht an Stub
  2. Marshalling/Serialisierung
  3. Übertragung der Parameter
  4. Ausführen auf dem Ziel
  5. Übertragung des Ergebnisses/Ausnahme
  6. Unmarshalling
  7. Rückgabe an den Aufrufer
- Optimierung für lokalen Fall

Wiedereintritt

- Wiedereintritt als Problem
- Vorkommen: Callbacks (von der unteren zur oberen Schicht) oder Multi-threading
- Problem: welche Funktionen dürfen benutzt werden
- Beispiel: Unix-Signal-Handler
- nicht mit Vor- und Nachbedingungen ausdrückbar

Fallstudie zur komponentenbasierten Entwicklung

Garlan u. a. (1995): Komposition eines Case-Tools aus

- einer objektorientierten Datenbank
- Toolkit zur Konstruktion graphischer Benutzeroberflächen
- event-basiertem Tool-Integrations-Mechanismus
- RPC-Mechanismus

Alle Komponenten in C oder C++ geschrieben.

Glaube und Wirklichkeit

Schätzung:

- Dauer: 6 Monate
- Aufwand: 12 PM

Tatsächlich:

- Dauer: 2 Jahre
- Aufwand: 60 PM für ersten Prototyp

Ergebnis:

- sehr großes System
- träge Performanz
- viele Anpassungen für die Integration notwendig
- existierende Funktionalität musste neu implementiert werden, weil sie nicht exakt den Anforderungen entsprach

Architektur-Mismatch

Definition

Architektur-Mismatch: inkompatible Annahmen von wiederzuverwendenden Komponenten über das Systems, in dem sie eingesetzt werden sollen.

Meist spät entdeckt, weil die Annahmen in aller Regel nur implizit sind.

Komponenten betreffend

Annahmen von Komponenten über

- ihre Infrastruktur, die zur Verfügung gestellt werden soll bzw. vorausgesetzt wird
  - Komponenten stellten vieles zur Verfügung, was gar nicht gebraucht wurde

  → exzessiver Code
- das Kontrollmodell: wer steuert den Kontrollfluss
  - jede Komponente nahm an, dass sie die Hauptkontrollschleife darstelle

  → aufwändige Restrukturierung notwendig
- das Datenmodell: Art und Weise, wie die Umgebung Daten manipuliert, die von der Komponente verwaltet werden
  - hierarchische Datenstruktur erlaubte Änderung der Teile nur über das Ganze

  → für Anwendung zu unflexibel

  → teilweise Neuimplementierung

Konnektoren betreffend

Annahmen von Konnektoren über

- Protokoll: Interaktionsmuster
  - Semantik des synchronen Aufrufs passte nicht
  - → Ausweichung auf RPC des Betriebssystems hierfür
- Datenmodell: Art der Daten, die kommuniziert werden
  - RPC des Betriebssystems nahm an, C-Datenstrukturen zu transportieren
  - wiederzuverwendenendes Event-Broadcast-System nahm an, ASCII zu transportieren
  - → Konvertierungsroutinen wurden notwendig

Architekturkonfiguration betreffend

Annahmen über Architekturkonfiguration

- Topologie der Systemkommunikation
  - Datenbank nahm an, dass verbundene Tools nicht kooperieren wollen und blockierte sie, um Sequenzialisierung zu garantieren
  - Tools mussten aber kooperieren
  - → eigener Transaktionsmonitor musste implementiert werden
- An- oder Abwesenheit bestimmter Komponenten und Konnektoren

Konstruktionsprozess betreffend

Annahmen über Konstruktionsprozess (wie Komponenten/Konnektoren aus generischen Einheiten erstellt werden – sowohl zur Übersetzungs- als auch zur Laufzeit)

Beispiele:

- Datenbank → Schema muss festgelegt werden
- Event-System → Menge der Ereignisse und Registration

Annahmen über die Reihenfolge, in der Teile erstellt und kombiniert werden.

- nicht zueinander passende Annahmen über den Konstruktionsprozess

→ Umwege machten Konstruktionsprozess aufwändig und kompliziert

Wiederholungs- und Vertiefungsfragen

- Was ist eine Komponente?
- Was ist die Idee der komponentenbasierten Entwicklung? Welche Ziele verfolgt sie?
- Diskutieren Sie Vor- und Nachteile maßgefertigter Software versus Software von der Stange.
- Was ist ein Komponentenmodell? Was von einem solchen üblicherweise festgelegt bzw. zur Verfügung gestellt?
- Was ist eine Schnittstelle und welche Bedeutung kommt diesem Konzept im Kontext komponentenbasierter Entwicklung zu?
- Wie können vorgefertigte Komponenten angepasst werden?
- Welches Problem tritt bei Anpassung durch Vererbung und Redefinition auf? Wie kann man mit ihm umgehen?
- Was ist das Problem des Wiedereintritts?
- Was versteht man unter Architektur-Mismatch und welche Bedeutung hat er für die komponentenbasierte Entwicklung?

Software-Architektur

6. Software-Architektur
   - Was ist Software-Architektur?
   - Zusammenfassung aus dem Software-Projekt: Hofmeister-Methode und -Blickwinkel
   - Qualität von Software-Architekturen
   - Taktiken
   - Evaluation von Software-Architektur
   - Wiederholungsfragen

Lernziele

- Verstehen, was Software-Architektur ist
- Qualitäten einer Architektur kennen
- Taktiken des Software-Architekturentwurfs kennen
- Software-Architektur beschreiben können
- Software-Architektur bewerten können

Was ist Architektur?

> **Architecture** *is the human organization of empty space using raw material.*
>
> Richard Hooker, 1996.

Definition

Software-Architektur ist die grundlegende Organisation eines Systems, verkörpert (IEEE P1471 2002)

- in seinen Komponenten,
- deren Beziehungen untereinander und zur Umgebung
- und die Prinzipien, die den Entwurf und die Evolution leiten.

Weitere über 100 Definitionen unter `www.sei.cmu.edu/architecture/community_definitions.html`.

Bedeutung von Software-Architektur

- Kommunikation zwischen allen Interessenten
  - hoher Abstraktionsgrad, der von vielen verstanden werden kann
- Frühe Entwurfsentscheidungen
  - → nachhaltige Auswirkungen
  - → frühzeitige Analyse
- Transferierbare Abstraktion des Systems
  - → Beherrschung der Komplexität
  - → Aufgabenverteilung
  - → eigenständig wiederverwendbar
  - → unterstützt Wiederverwendung im großen Stil (Software-Produktlinien)

Bedeutung von Software-Architektur

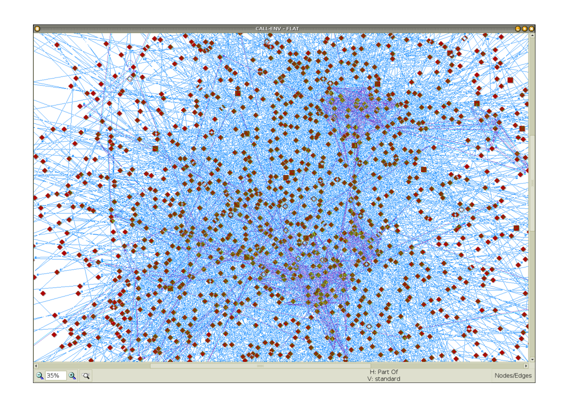
- Kommunikation zwischen allen Interessenten
  - hoher Abstraktionsgrad, der von vielen verstanden werden kann
- Frühe Entwurfsentscheidungen
  - → nachhaltige Auswirkungen
  - → frühzeitige Analyse
- Transferierbare Abstraktion des Systems
  - → Beherrschung der Komplexität
  - → Aufgabenverteilung
  - → eigenständig wiederverwendbar
  - → unterstützt Wiederverwendung im großen Stil (Software-Produktlinien)

- SA repräsentiert hohe Abstraktion eines Systems, die von den meisten Interessenten verstanden werden kann und dam eine Grundlage zum gegenseitigen Verständnis, zur Konsensbildung und zur Kommunikation darstellt.
- SA ist die Manifestation früher Entwurfsentscheidungen; diese frühe Fixierung kann nachhaltige Auswirkungen haben auf die nachfolgende Entwicklung, Auslieferung sowie Wartung und Evolution. SA ist auch die früheste Systembeschreibung, die analysiert werden kann.
- SA konstituiert ein relativ kleines intellektuell fassbares Modell darüber, wie das System strukturiert ist und wie seine Komponenten zusammenwirken; dieses Modell ist eigenständig nutzbar und kann über das spezifische System hinaus transferiert werden; insbesondere kann es für Systeme mit ähnlichen Eigenschaften und Anforderungen wiederverwend werden, um so Wiederverwendung im großen Stil zu unterstützen (Stichwort: Software-Produktlinien).

Architektursichten und -blickwinkel (IEEE P1471 2002)

Definition

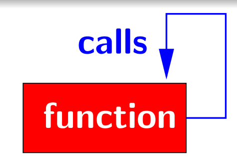
Architektursicht (View): Repräsentation eines ganzen Systems aus der Perspektive einer kohärenten Menge von Anliegen.

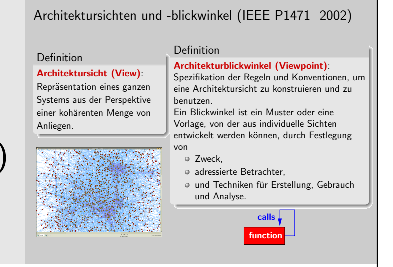


Definition

Architekturblickwinkel (Viewpoint): Spezifikation der Regeln und Konventionen, um eine Architektursicht zu konstruieren und zu benutzen.
Ein Blickwinkel ist ein Muster oder eine Vorlage, von der aus individuelle Sichten entwickelt werden können, durch Festlegung von

- Zweck,
- adressierte Betrachter,
- und Techniken für Erstellung, Gebrauch und Analyse.



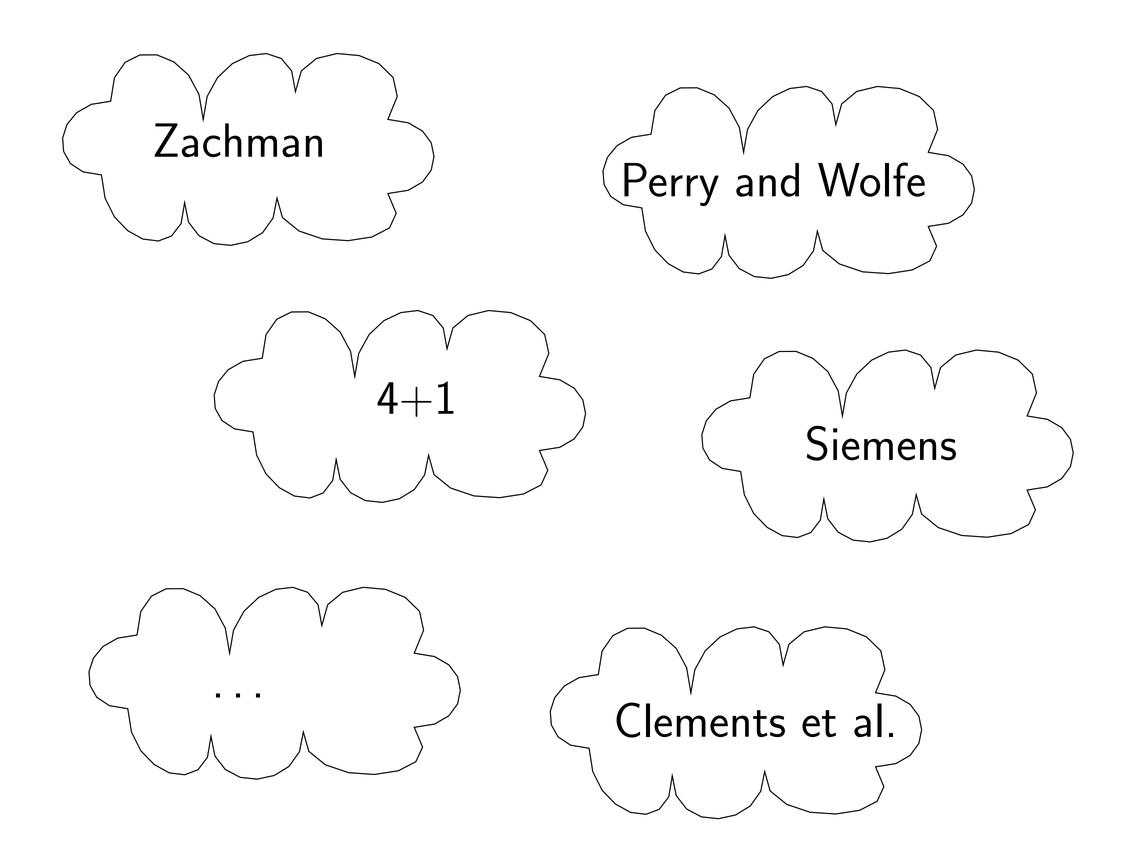


Unterschiedliche Sichten helfen der Strukturierung: Separation of Concerns.
Architecture design and reconstruction create architectural views for existing systems. But what is a view at all?
One of the achievements of the IEEE P1471 is the definition of views and viewpoints.
A *view* is a representation of a whole system from the perspective of a related set of concerns. Here, for instance, you see a part of the call graph of jikes, the IBM compiler for Java.
Such views are formalized through viewpoints. A *viewpoint* specifies the kind of information that can be put in a view. A call graph viewpoint can be modeled by this UML diagram, for instance.

Siemens-Blickwinkel (Hofmeister u. a. 2000)

- **Konzeptioneller Blickwinkel**: beschreibt logische Struktur des Systems; abstrahiert weitgehend von technologischen Details
- **Modulblickwinkel**: beschreibt die statische logische Struktur des Systems
- **Ausführungsblickwinkel**: beschreibt die dynamische logische Struktur des Systems
- **Code-Blickwinkel**: beschreibt die „anfassbaren" Elemente des Systems (Quelldateien, Bibliotheken, ausführbare Dateien etc.)

Verbreitete Blickwinkel

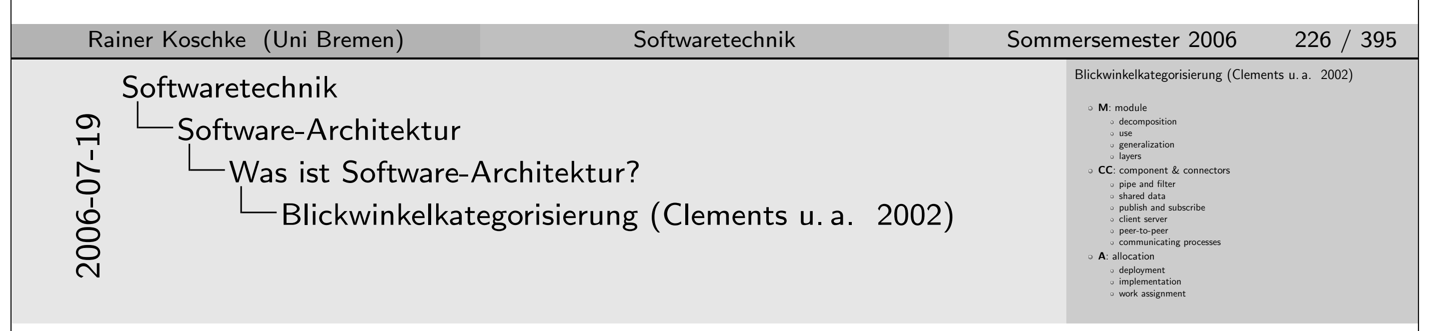


Verbreitete Blickwinkel
Zachman
Perry and Wolfe
4+1
Siemens
...
Clements et al.

Viewpoints are very popular in forward engineering. You likely know these. Zachman was one of the first authors on viewpoints. He proposed 6×6 different viewpoints. Perry and Wolfe proposed a simplified version of these views, distinguishing only three viewpoints. Then you have the 4+1 viewpoints by Philippe Kruchten, you have the four Siemens viewpoints, et cetera.
The number of viewpoints is confusing, in particular, because many of them are very similar.
Recently, the book by Clements and colleagues brought some order to this sea of viewpoints.

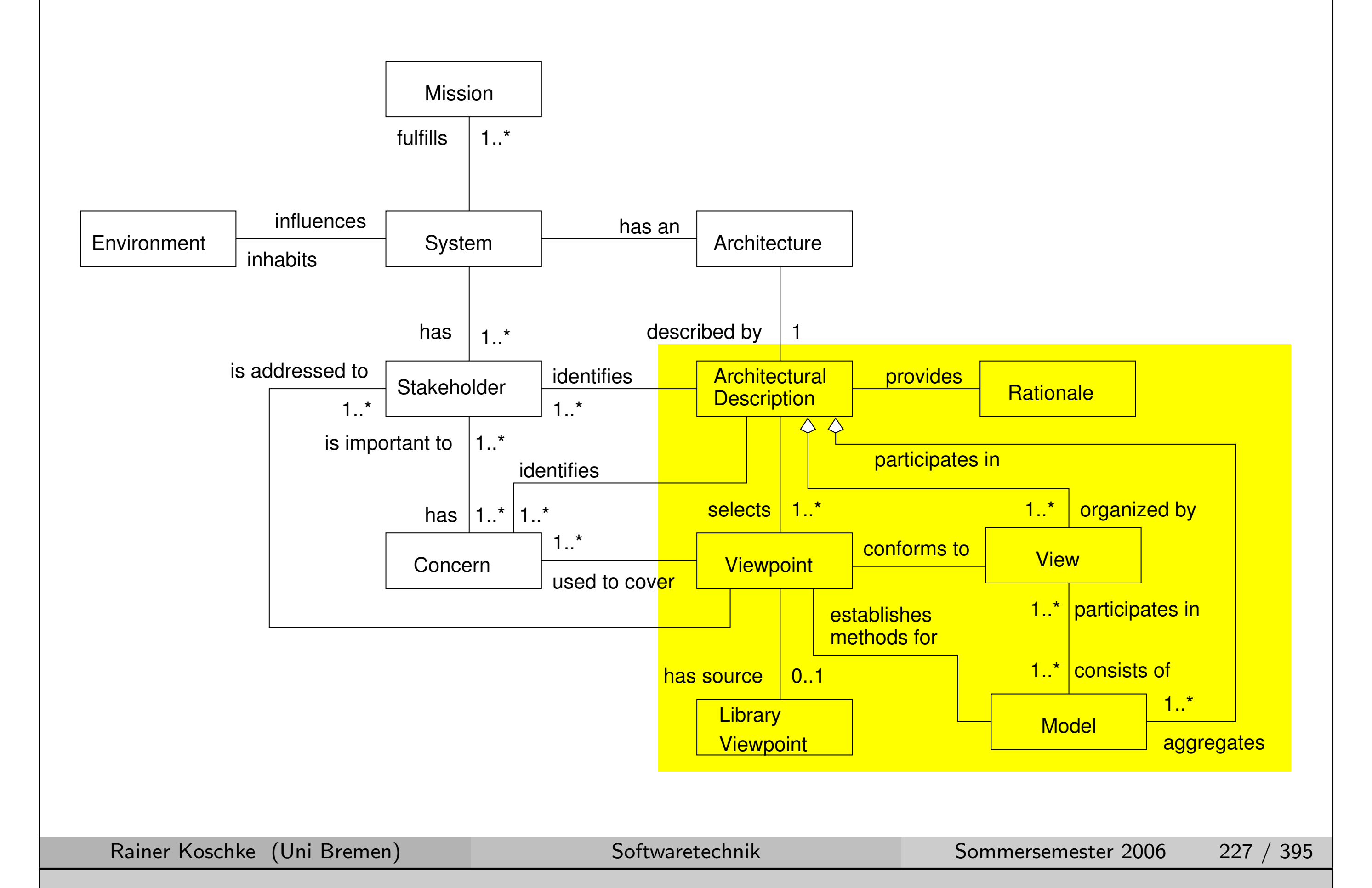
Blickwinkelkategorisierung (Clements u. a. 2002)

- **M**: module
  - decomposition
  - use
  - generalization
  - layers
- **CC**: component & connectors
  - pipe and filter
  - shared data
  - publish and subscribe
  - client server
  - peer-to-peer
  - communicating processes
- **A**: allocation
  - deployment
  - implementation
  - work assignment



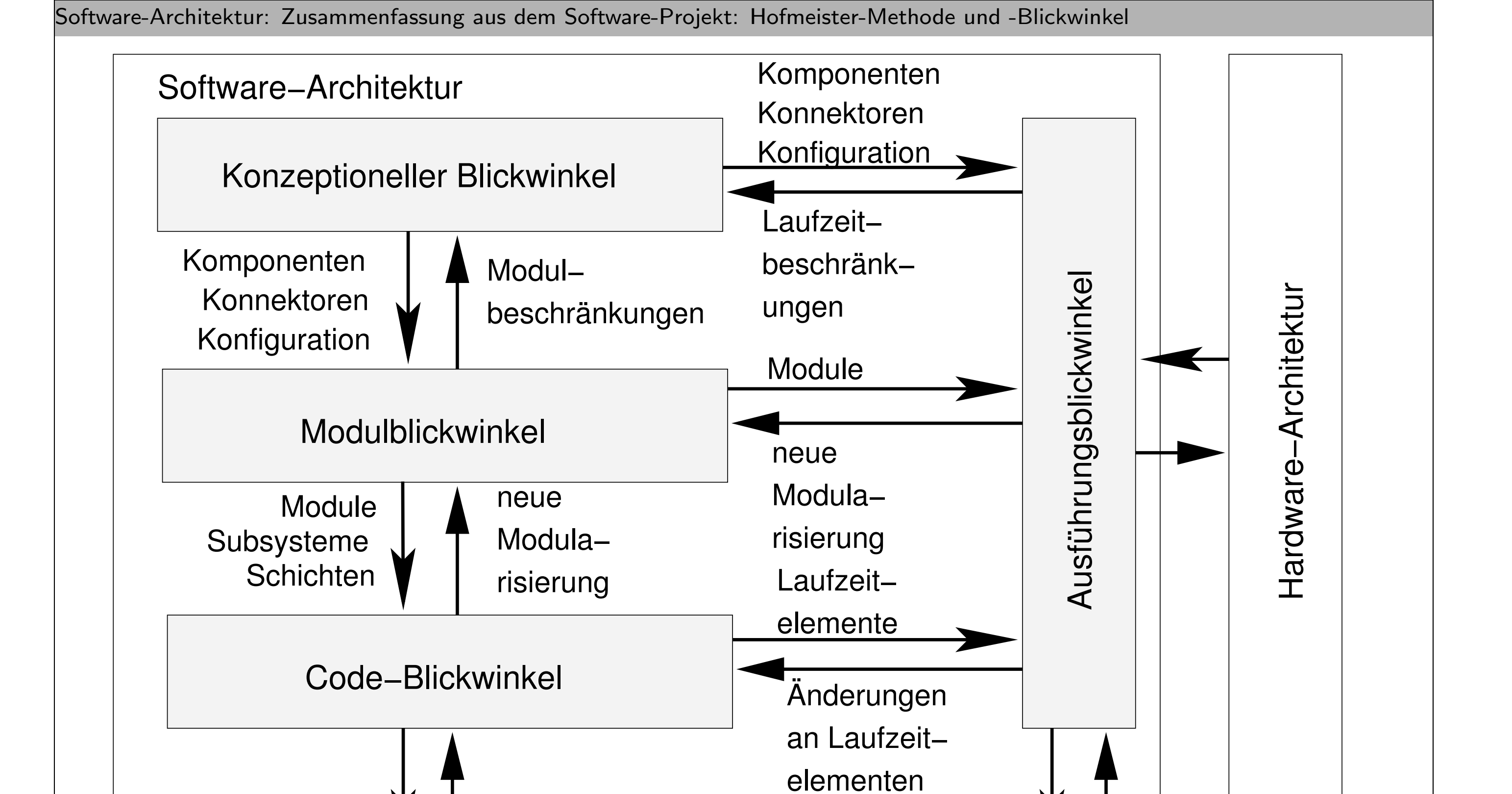
Here, you see their categories of viewpoints.
Module viewpoints show static structure and describe the decomposition, layering, and generalization of modules and their use dependencies. A module is a code unit that implements a set of responsibilities.
Component-and-connector viewpoints express runtime behavior described in terms of components and connectors. A component is one of the principal processing units of the executing system; a connector is an interaction mechanism for the components.
Allocation viewpoints describe mappings of software units to elements of the environment (the hardware, the file systems, or the development team).

Architekturbegriffe im Kontext (IEEE P1471 2002)



Methode von Hofmeister u. a. (2000)

1. Einflussfaktoren identifizieren
   - Produktfunktionen und -attribute
   - technologische Faktoren
   - organisatorische Faktoren
2. konkurrierende Faktoren feststellen
3. Kompromisse für Faktorenkonflikte durch Strategien finden
4. iterativer Entwurf der verschiedenen Sichten



Funktionalität versus Qualität

Definition

Funktionalität: Umsetzung der funktionalen Anforderungen; die Fähigkeit eines Software-Systems, auf eine Eingabe die erwartete Ausgabe zu produzieren.

Funktionalität kann durch beliebige Strukturen umgesetzt werden; ist damit weitgehend unabhängig von Struktur.

Software-Architektur schränkt mögliche Strukturen ein aufgrund anderer Qualitätsattribute.

Definition

Qualität ist der Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt.

EN ISO 9000:2005

Beispiele für Software-Qualitätsaspekte

- Änderbarkeit
- Testbarkeit
- Sicherheit
- Robustheit
- Gebrauchstauglichkeit (Usability)
- Performanz
- Verfügbarkeit
- Skalierbarkeit
- Portierbarkeit
- ...

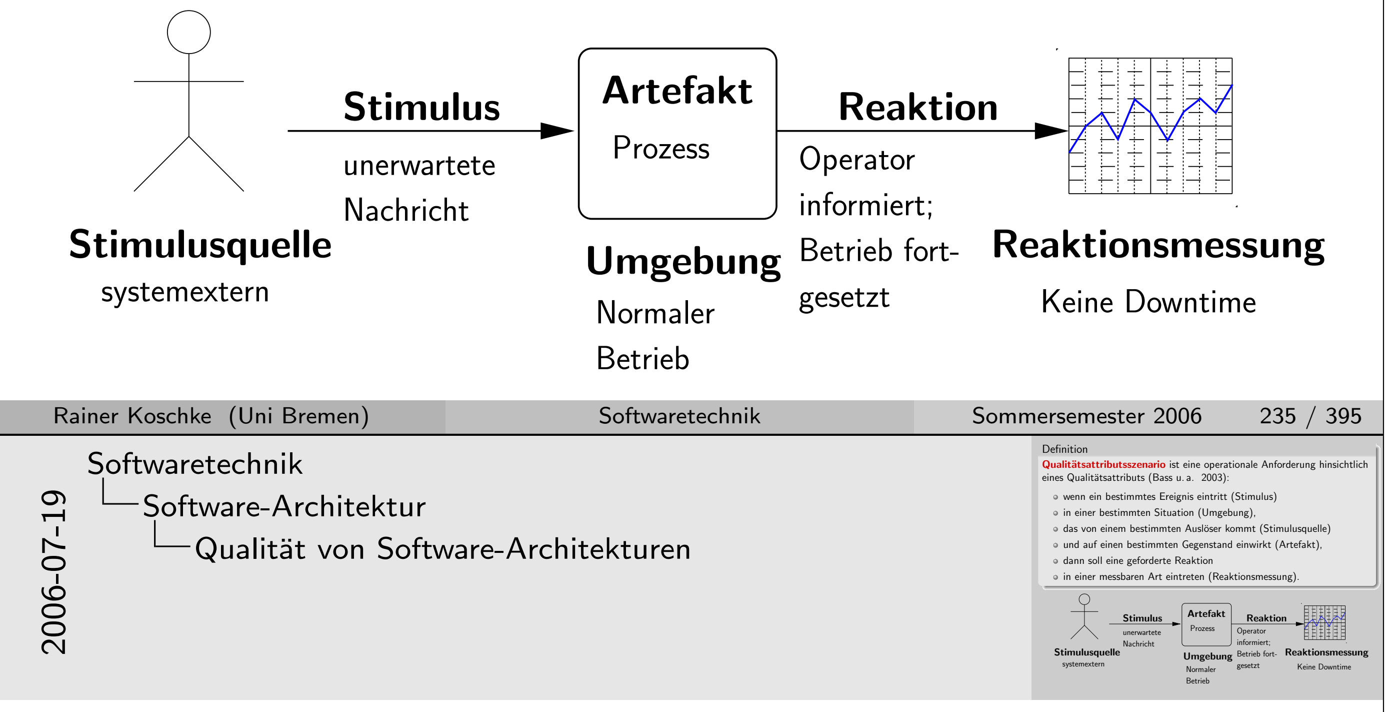
Software-Architektur und Qualitätsattribute

- Architektur ist kritisch für die Realisierung vieler Qualitäten
  - → Qualitäten müssen durch Konstruktion eingebaut werden
  - → Qualitäten können und sollen auf Architekturebene evaluiert werden
- Architektur alleine genügt nicht zur Erreichung der Qualitäten
  - → Architektur bildet nur die Grundlage
  - → Implementierungsdetails sind maßgebend
- Qualitätsattribute können im Konflikt zueinander stehen; Architektur ist ein Kompromiss
- Qualitätsattribute müssen objektiv und operational beschrieben sein
  → konkrete messbare Szenarien

Definition

Qualitätsattributsszenario ist eine operationale Anforderung hinsichtlich eines Qualitätsattributs (Bass u. a. 2003):

- wenn ein bestimmtes Ereignis eintritt (Stimulus)
- in einer bestimmten Situation (Umgebung),
- das von einem bestimmten Auslöser kommt (Stimulusquelle)
- und auf einen bestimmten Gegenstand einwirkt (Artefakt),
- dann soll eine geforderte Reaktion
- in einer messbaren Art eintreten (Reaktionsmessung).



Beispiel für Verfügbarkeit: Erkennung von Fehlern mit Ausnahmebehandlung für volle Fehlertoleranz.

Kategorien von Software-Architektur-Qualitäten

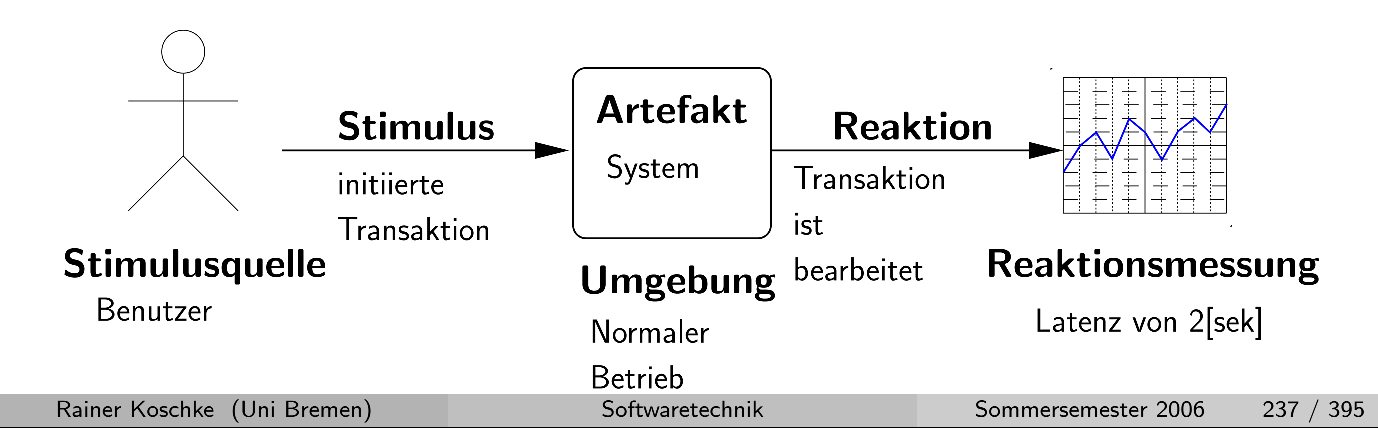
- Systemqualitäten (Verfügbarkeit, Änderbarkeit, Performanz, Sicherheit, Testbarkeit, Gebrauchstauglichkeit etc.)
- Geschäftsqualitäten, z.B. Time-To-Market
- Qualitäten der Architektur selbst, z.B. konzeptionelle Integrität, die indirekt die anderen Qualitäten beeinflussen

Performanz

Allgemeine Szenarien:

Quelle	intern/extern
Stimulus	periodische, sporadisch, stochastische Ereignisse
Artefakt	System
Umgebung	Normalbetrieb, ausgelastet
Reaktion	Bearbeitung von Stimuli, Änderung von Service-Levels
Maß	Latenz, Deadline, Durchsatz, Versatz, Versäumnisrate, Datenverlust

Spezielles Szenario:



Performanz

Allgemeine Szenarien:

Quelle	intern/extern
Stimulus	periodische, sporadisch, stochastische Ereignisse
Artefakt	System
Umgebung	Normalbetrieb, ausgelastet
Reaktion	Bearbeitung von Stimuli, Änderung von Service-Levels
Maß	Latenz, Deadline, Durchsatz, Versatz, Versäumnisrate, Datenverlust

Spezielles Szenario:

Stimulusquelle: Benutzer — Stimulus: initiierte Transaktion — Artefakt: System — Umgebung: Normaler Betrieb — Reaktion: Transaktion ist bearbeitet — Reaktionsmessung: Latenz von 2[sek]

Latenz:	Reaktionszeit gemessen ab Eintreffen der Nachricht (Latency)
Deadline:	fester Zeitpunkt, zu dem Reaktion erfolgt sein muss
Versatz:	Variation der Reaktionszeit (Jitter)
Durchsatz:	Anzahl der Ereignisse, die einem bestimmten Zeitintervall bearbeitet werden können
Versäumnisrate:	Anzahl der Ereignisse, die einem bestimmten Zeitintervall nicht bearbeitet werden konnten
Datenverlust:	Umfang der Daten, die verloren gingen

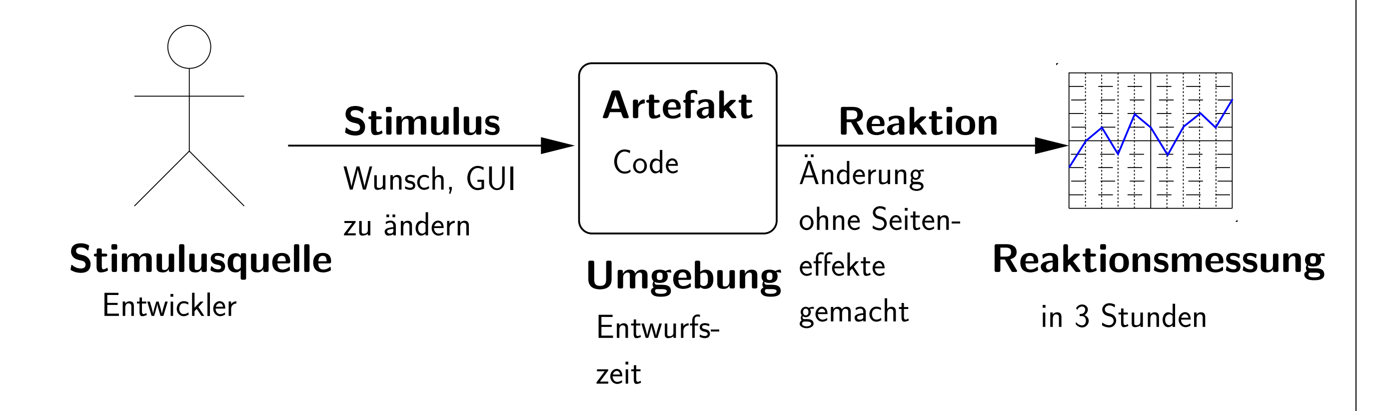
Änderbarkeit

Allgemeine Szenarien:

Quelle	Endbenutzer, Entwickler, Systemadministrator
Stimulus	Wunsch, Funktionalität hinzuzufügen, zu entfernen, abzuändern, zu variieren bzw. Qualitätsaspekt zu verändern
Artefakt	System (Benutzeroberfläche, Plattform, Umgebung, kooperierendes System)
Umgebung	Laufzeit, Ladezeit, Übersetzungszeit, Entwurfszeit
Reaktion	Lokalisierung, Änderung, Test, Auslieferung der Architekturkomponenten
Maß	Kosten in Form von Anzahl der betroffenen Komponenten, Aufwand, Geld; Ausmaß des Einflusses auf andere Qualitätsattribute

Änderbarkeit

Spezielles Szenario:



Testbarkeit

Definition

Testbarkeit: Grad der Einfachheit, Fehler in der Software aufzuzeigen; Wahrscheinlichkeit – unter der Voraussetzung, dass die Software mindestens einen Fehler hat – dass der nächste Test positiv ausfällt.

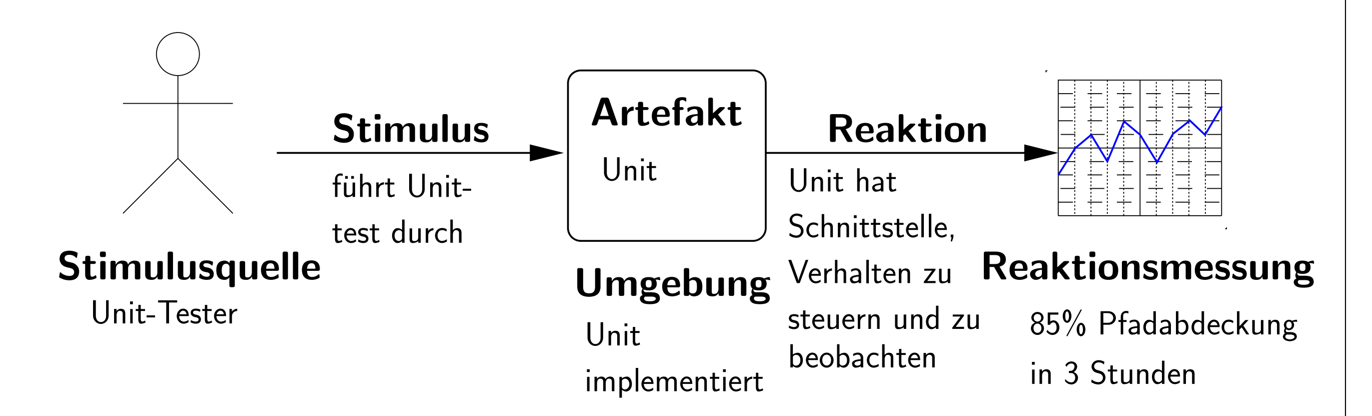
Testbarkeit

Allgemeine Szenarien:

Quelle	Unit-Entwickler, Integrator, Systemverifizierer, Akzeptanztester, Endbenutzer
Stimulus	Prüfling (Analyse, Architektur, Entwurf, Klasse, Subsystem) erstellt; System ausgeliefert
Artefakt	Teil des Enwurfs oder Codes; ganze Applikation
Umgebung	Entwurfs-, Entwicklungs-, Übersetzungs-, Einsatzzeit
Reaktion	gewährt Einblick in Zustandswerte und berechnete Werte, bereitet Testumgebung vor
Maß	Abdeckungsgrad, Wahrscheinlichkeit eines Störfalls, wenn ein Fehler existiert; Aufwand für Test, Dauer, Vorbereitung der Testumgebung

Testbarkeit

Spezielles Szenario:



Geschäftsqualitäten

- Time-To-Market
- Kosten/Nutzen
- anvisierte Lebensdauer
- Zielmarkt
- Auslieferungsplan
- Integration mit Legacy-Systemen
- ...



- Time-To-Market: kurze Time-To-Markt zwingt zu COTS und inkrementeller Entwicklung
- Kosten/Nutzen: hat z.B. auch Einfluss auf Technologien, die verwendet werden können (Anschaffungskosten, Einarbeitungszeit)
- anvisierte Lebensdauer: hohe Lebensdauer verstärkt Bedeutung von Änderbarkeit, Skalierbarkeit, Portabilität
- Zielmarkt: bei Massenmarkt ist Portabilität und allgemeine Funktionalität von hoher Bedeutung
- Auslieferungsplan: bei inkrementeller Auslieferung muss das System leicht erweiterbar sein
- Integration mit Legacy-Systemen: zwingt zu Integrationsmechanismen

Qualitäten der Architektur selbst

- Einfachheit
- konzeptionelle Integrität: Gleiches wird gleich gelöst (Fred Brooks)
- Kopplung minimieren, Kohäsion maximieren
- Isomporphie zur Realität (Michael Jackson)
- ...

Taktiken und Strategien

Definition

Taktik bezeichnet das geschickte Nutzen einer gegebenen Lage (Wikipedia).

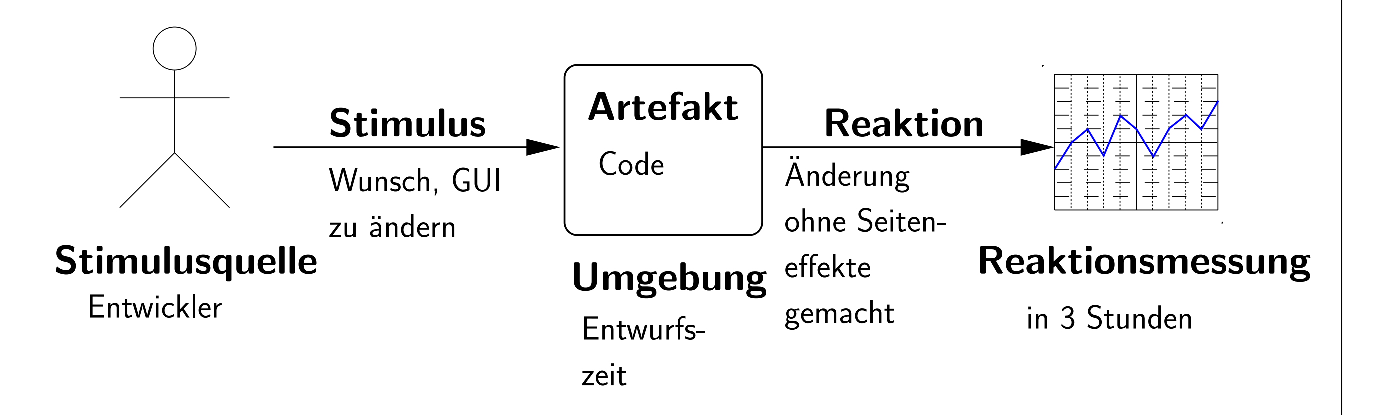
Strategisches Handeln ist langfristig, taktisches Handeln mittelfristig und operatives Handeln kurzfristig angelegt (Wikipedia).

"Tactics are the specifics of strategies".

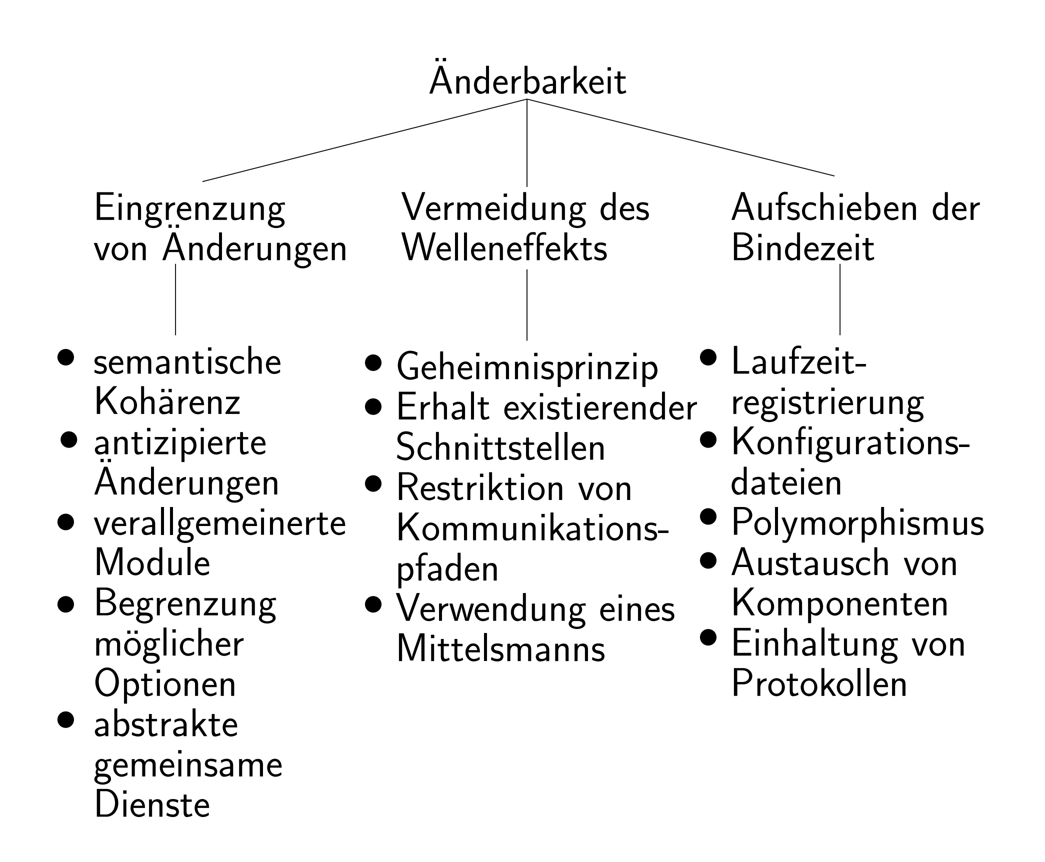
Taktik im Architekturkontext

Definition

Taktik: Entwurfsentscheidung, die die Reaktion auf einen Stimulus bestimmt und damit ein Qualitätsattribut beeinflusst.



Taktiken für Änderbarkeit





Eingrenzung von Änderungen

semantische Kohärenz: alle Bestandteile eines Moduls tragen zu einem gemeinsamen Zweck bei; bei Änderung des Zwecks müssen nur die Elemente des Moduls angepasst werden

antizipierte Änderungen: zukünftige Änderungen sind bereits eingeplant/eingebaut

verallgemeinerte Module: Modul ist allgemeiner als es sein müsste und parametrisiert (durch einfache Parameter bis hin zur Ausprägung des Moduls als Interpreter)

Begrenzung möglicher Optionen: die möglichen Optionen werden eingeschränkt; damit werden beliebige Änderungen ausgeschlossen; z.B. kann man bei Änderungen des Prozessors einschränken, dass er sich nur innerhalb einer Prozessorfamilie ändern darf

abstrakte gemeinsame Dienste: ähnliche Funktionalität in verschiedenen Modulen wird herausfaktorisiert und nur einmal implementiert als dienstleistendes Modul

Vermeidung des Welleneffekts

Geheimnisprinzip: Dinge, die sich wahrscheinlich ändern, werden hinter einer abstrakten Schnittstelle verborgen.

Erhalt existierender Schnittstellen:

- mehrere Schnittstellen: neu und alt
- Adapter
- Stumpf (wenn Dienst wegfällt)

Restriktion von Kommunikationspfaden: Anzahl der Module, mit denen Daten geteilt werden, wird reduziert

Verwendung eines Mittelsmanns I

Verwender (V) und Dienstleister (D)

- Syntax von ...
  - Daten: Format von Daten zwischen V und D

  → Verwendung eines Repositories, das Daten konvertiert
  - Diensten: Signaturen müssen übereinstimmen

  → Muster: Facade, Mediator, Strategy, Proxy, Factory
- Semantik von ...
  - Daten: konsistente Annahmen über Semantik der Daten
  - Diensten: konsistente Annahmen über Semantik der Dienste

→ semantischer Konverter

- Reihenfolge von...
  - Daten: konsistente Annahme über Reihenfolge
  - Kontrolle: D muss vor V ausgeführt sein (in bestimmter Zeit)

→ Puffer mit Veränderung der Reihenfolge

Verwendung eines Mittelsmanns II

- Identität der Schnittstelle von D (falls es mehrere gibt)

→ Broker-Muster

- Ort von D (zur Laufzeit); z.B. gleicher Prozessor

→ Name-Server

- Quality-of-Service/-Data von D

→ schwer zu überbrücken

- Existenz von D

→ Factory-Muster

- Ressourcenverhalten von D

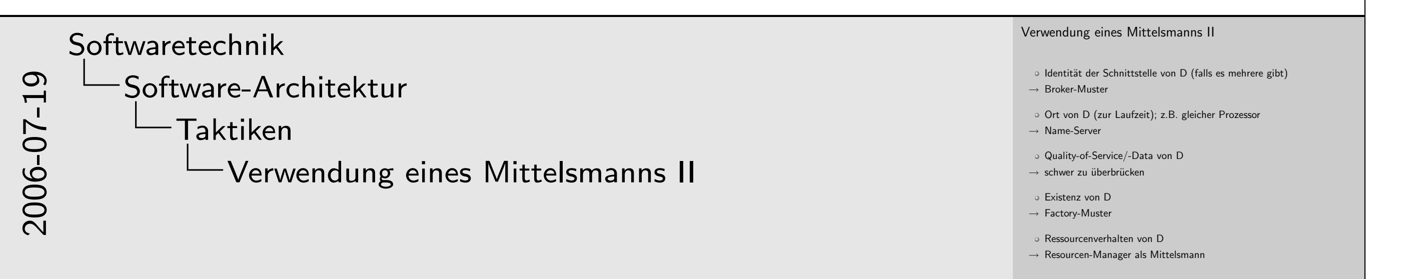
→ Resourcen-Manager als Mittelsmann

Verwendung eines Mittelsmanns II

- Identität der Schnittstelle von D (falls es mehrere gibt)
→ Broker-Muster
- Ort von D (zur Laufzeit); z.B. gleicher Prozessor
→ Name-Server
- Quality-of-Service/-Data von D
→ schwer zu überbrücken
- Existenz von D
→ Factory-Muster
- Ressourcenverhalten von D
→ Resourcen-Manager als Mittelsmann

Beispiele:

- Syntax von ...
  - Daten: die Sequenz von Bytes (Big vs. Little Endian)
  - Diensten: Anzahl und Typ der Parameter
- Semantik von ...
  - Daten: die Einheit der Werte ist die gleiche; Meilen versus Kilometer (alles schon 'mal dagewesen)
  - Diensten: `top` wirft Exception, wenn der Stack leer ist statt einen undefinierten Wert zurückzugeben

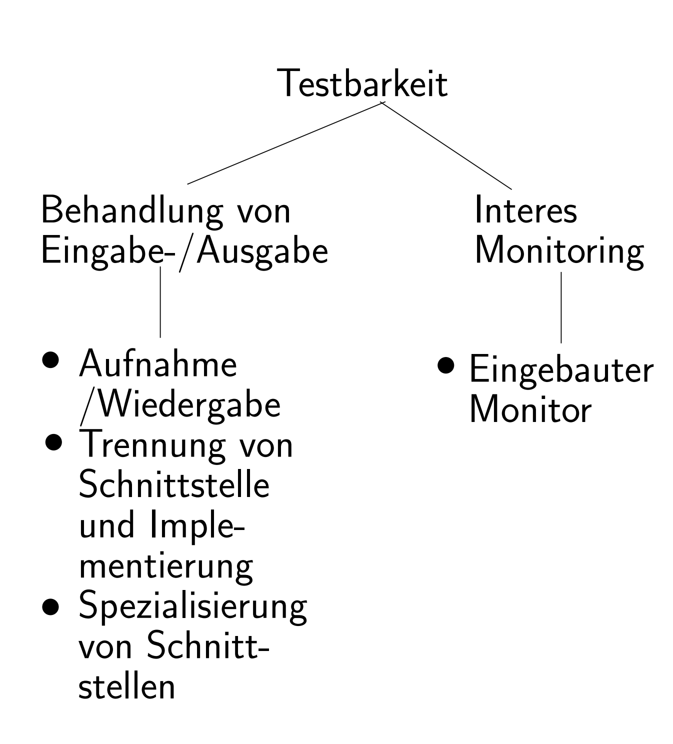


- Reihenfolge von ...
  - Daten: Netzwerkpakete kommen in der Reihenfolge ihres Abschickens an
  - Kontrolle: D muss 5 Millisekunden vor V ausgeführt worden sein
- Identität der Schnittstelle von D: D hat verschiedene Versionen seiner Schnittstelle über die Zeit
- Ort von D (zur Laufzeit); V und D laufen auf gleichem Prozessor mit gemeinsamem Speicher
- Quality-of-Service/-Data von D: die Daten des Sensors D sind in einem Toleranzbereich der Genauigkeit
- Existenz von D: damit V einen Dienst von D aufrufen kann, muss D existieren oder V muss die Möglichkeit haben, D erschaffen
- Ressourcenverhalten von D: D gibt alle seine Resourcen wieder frei, nachdem der Dienst erbracht wurde

Aufschieben der Bindezeit

- Laufzeitregistrierung: Plug-and-Play zur Laufzeit oder Ladezeit
- Konfigurationsdateien: Parameterwerte während des Systemstarts
- Polymorphismus: spätes Binden von Methoden
- Austausch von Komponenten: während der Ladezeit
- Einhaltung von Protokollen: Laufzeit-Bindung unabhängiger Prozesse

Taktiken für Testbarkeit



Behandlung von Eingabe und Ausgabe

Aufnahme/Wiedergabe

- Information, die Schnittstelle passiert, wird vermerkt
- kann später für Regressionstest benutzt werden

Trennung von Schnittstelle und Implementierung

- ermöglicht Substitution der Implementierung fürs Testen
- Teststümpfe können vorausgesetzte Komponenten simulieren
- Testtreiber simulieren Verwender

Spezialisierung von Zugriffspfaden/Schnittstellen

- Spezialisierte Schnittstelle erlaubt Aufzeichnung und Manipulation von Attributen einer Komponente durch einen Testrahmen

Internes Monitoring

Eingebauter Monitor

- Zustand und andere Attribute (Performanz, Belastung (Load), Sicherheit etc.) werden durch Schnittstelle zur Verfügung gestellt
- wird über Schnittstelle vom Monitor zur Laufzeit beobachtet
- Permanente Schnittstelle: Teil der normalen Schnittstelle
- Temporäre Schnittstelle: nur beim Monitoring präsent (Ausgabeanweisungen im Code fürs Tracing, Code-Instrumentierung, Makros, aspektorientiertes Programmieren etc.)

Kosten von Evaluationen

Erfahrungen bei AT&T:

- ca. 300 Architekur-Reviews durchgeführt
- für Projekte mit mindestens 700 PT Aufwand

$\rightarrow$ durchschnittlich 70 PT für Evaluation

Erfahrungen des SEIs:

- 36 PT für ATAM-Evaluation (nur Evaluations-Team; dazu noch: andere Projektteilnehmer und Entscheider)

Vorteile einer frühen Evaluation

- frühe Erkennung von Fehlern (je früher ein Fehler entdeckt wird, desto billiger ist seine Beseitigung)
  $\rightarrow$ 10% Kosteneinsparung bei AT&T
- Zwang zur Dokumentation der Architektur
- Zwang zum Festhalten der Begründungen von Entwurfsentscheidungen
- weitere Überprüfung der Anforderungen
- Verbesserung von Architekturen durch Erfahrungen, die man in den Evaluationen sammelt

Techniken zur Evaluation von Software-Architektur

- Fragetechniken
  - Fragebögen und Checklisten
  - Architecture Tradeoff Analysis Method (ATAM)
  - Cost Benefit Analysis Method (CBAM)
- Messtechniken
  - Architekturmetriken (Kopplung, Kohäsion etc.)
  - Simulatoren (Performanz, Verfügbarkeit)

→ Fragetechniken sind jederzeit anwendbar, aber weniger objektiv

→ Messtechniken setzen Architektur voraus, liefern aber genaue Antworten

Anforderungen/Kontext

Architekturanalysen sind schwierig:

- große Systeme haben umfangreiche Architektur
- Evaluation muss Verbindung zu Geschäftszielen herstellen
- verschiedene Stakeholders haben unterschiedliche Interessen

Architecture Tradeoff Analysis Method (ATAM)

Vorbedingungen:

- klar artikulierte Ziele und Anforderungen an die Architektur
- klar abgesteckter Rahmen (ca. fünf Ziele mit hoher Priorität)
- erwarteter Nutzen übersteigt erwartete Kosten (meist für Systeme ab mittlerer Größe erfüllt)
- Schlüsselrollen verfügbar (z.B. Architekt)
- kompetentes Evaluations-Team (Querschnittsbereich mit Entscheidungskompetenzen)
- realistische und offene Erwartungen

Teilnehmer bei ATAM (Bass u. a. 2003)

- Evaluations-Team (extern)
- Entscheider
  - Projekt-Manager
  - Architekt
  - eventuell Vertreter des Kunden
- Architektur-Stakeholders
  - Entwickler, Tester, Integrierer, Wartungsprogrammierer, Performanztuner, Benutzer, Build-Prozess-Veranwortliche

Rollen im ATAM-Evaluations-Team

Gruppenleiter



Aufgaben:

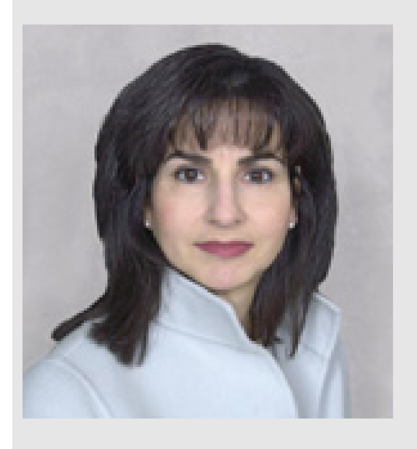
- bereitet Evaluation vor
- hält Kontakt zum Kunden
- formiert Evaluations-Team
- stellt sicher, dass Endbericht ausgeliefert wird

Eigenschaften:

- Organisationsgabe
- Managementfähigkeiten
- gute Interaktion mit Kunden
- zuverlässig in Zeitplänen

Rollen im ATAM-Evaluations-Team

Evaluationsleiterin



Aufgaben:

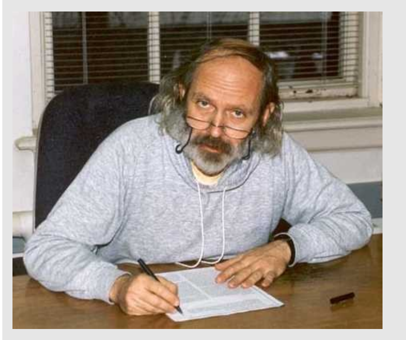
- leitet Evaluation
- unterstützt Auswahl der Szenarien
- organisiert Szenarioauswahl und -priorisierung
- unterstützt Evaluation

Eigenschaften:

- erfahren in Architektur
- kann präsentieren
- kann moderieren

Rollen im ATAM-Evaluations-Team

Szenario-Schreiber



Aufgaben:

- hält Szenarien fest während der Auswahl (Flipchart o.Ä.)
- hält Terminologie fest
- besteht auf klarer Formulierung

Eigenschaften:

- besteht darauf, die Dinge auf den Punkt zu bringen
- kann die Essenz einer Diskussion aufnehmen und sie prägnant zusammenfassen

Rollen im ATAM-Evaluations-Team

Protokollant





Aufgaben:



- protokolliert initiale Szenarien sowie Motivation und Entschluss für Szenarien
- verschickt Szenarien an alle Beteiligten

Eigenschaften:

- kann sich schriftlich gut ausdrücken
- kann Information gut abrufen
- hat gutes Verständnis von Architekturfragen
- kann technische Aspekte gut aufnehmen
- ist bereit, Diskussion zu unterbrechen, um sein eigenes Verständnis eines Szenarios zu prüfen

Rollen im ATAM-Evaluations-Team

Zeitnehmerin



Aufgaben:

- unterstützt Evaluationsleiter, die Zeit einzuhalten
- unterstützt, die Zeit für jedes Szenario in der Evaluationsphase festzuhalten und zu steuern

Eigenschaften:

- bereit, Diskussion mit Hinweis auf Zeit zu unterbrechen

Rollen im ATAM-Evaluations-Team

Prozessbeobachterin

Aufgaben:



- entdeckt Verbesserungen des Evaluationsprozesses
- eher stille Beobachtung während der Evaluation, kann aber Prozessvorschläge machen
- berichtet Erfahrungen und schlägt Verbesserung nach der Evaluation vor
- berichtet an unternehmensweite Architekturgruppe



Eigenschaften:

- guter Beobachter
- erfahrener Anwender der Methode

Rollen im ATAM-Evaluations-Team

Prozessexperte (Process Enforcer)

Aufgaben:

- unterstützt Evaluationsleiter, die Prozessschritte richtig auszuführen

Eigenschaften:

- erfahrener Anwender der Methode
- diskreter Ratgeber

Rollen im ATAM-Evaluations-Team

Fragesteller





Aufgaben:

- wirft Aspekte auf, an die die Stakeholders nicht gedacht haben

Eigenschaften:

- hat fundiertes Architekturwissen
- hat Einsicht in die Belange der Stakeholders
- hat Erfahrung mit Systemen in ähnlichen Domänen
- ohne Angst, auch umstrittene Belange aufzuwerfen

Resultate von ATAM

Primäre Resultate:

- präzise Beschreibung der Architektur
- Artikulation der Geschäftsziele
- Qualitätsanforderungen in Form von Szenarien
- Verbindung von Entwurfsentscheidungen und Qualitätsanforderungen
- Liste von Einflüssen (Sensitivity Points) und Kompromissen (Tradeoff Points)
- Liste von Risiken (Risks) und Nichtrisiken (Nonrisks)

Einflüsse/Kompromisse

Definition

Sensitivity Point: Entwurfsentscheidung mit merklichem Einfluss auf ein Qualitätsattribut

Bsp.: *Backup-Datenbank soll verwendet werden*
→ (positiver) Einfluss auf Zuverlässigkeit

Definition

Tradeoff Point: Sensitivity Point mit merklichem Einfluss auf mehrere Qualitätsattribute

Bsp.: *Backup-Datenbank soll verwendet werden*
→ (negativer) Einfluss auch auf Performanz, wegen zusätzlichen Ressourcenbedarfs

Risiken/Nichtrisiken

Definition

Risiko: Entwurfsentscheidung mit potentiell unerwünschten Konsequenzen für die Qualitätsattribute.
Nichtrisiko: Entwurfsentscheidung, die nach Analyse keine unerwünschten Konsequenzen für die Qualitätsattribute hat.

Bsp.: *Backup-Datenbank soll verwendet werden*
→ (positiver) Einfluss auf Zuverlässigkeit
→ (negativer) Einfluss auch auf Performanz

Wird zum Risiko, erst wenn Performanz von großer Bedeutung ist.

Resultate von ATAM (Forts.)

Erwünschte Nebeneffekte:

- bessere Architekturbeschreibung
- Gemeinschaftsgefühl aller Beteiligter
- bessere und offenere Kommunikation aller Beteiligter
- besseres Verständnis der jeweiligen Anliegen

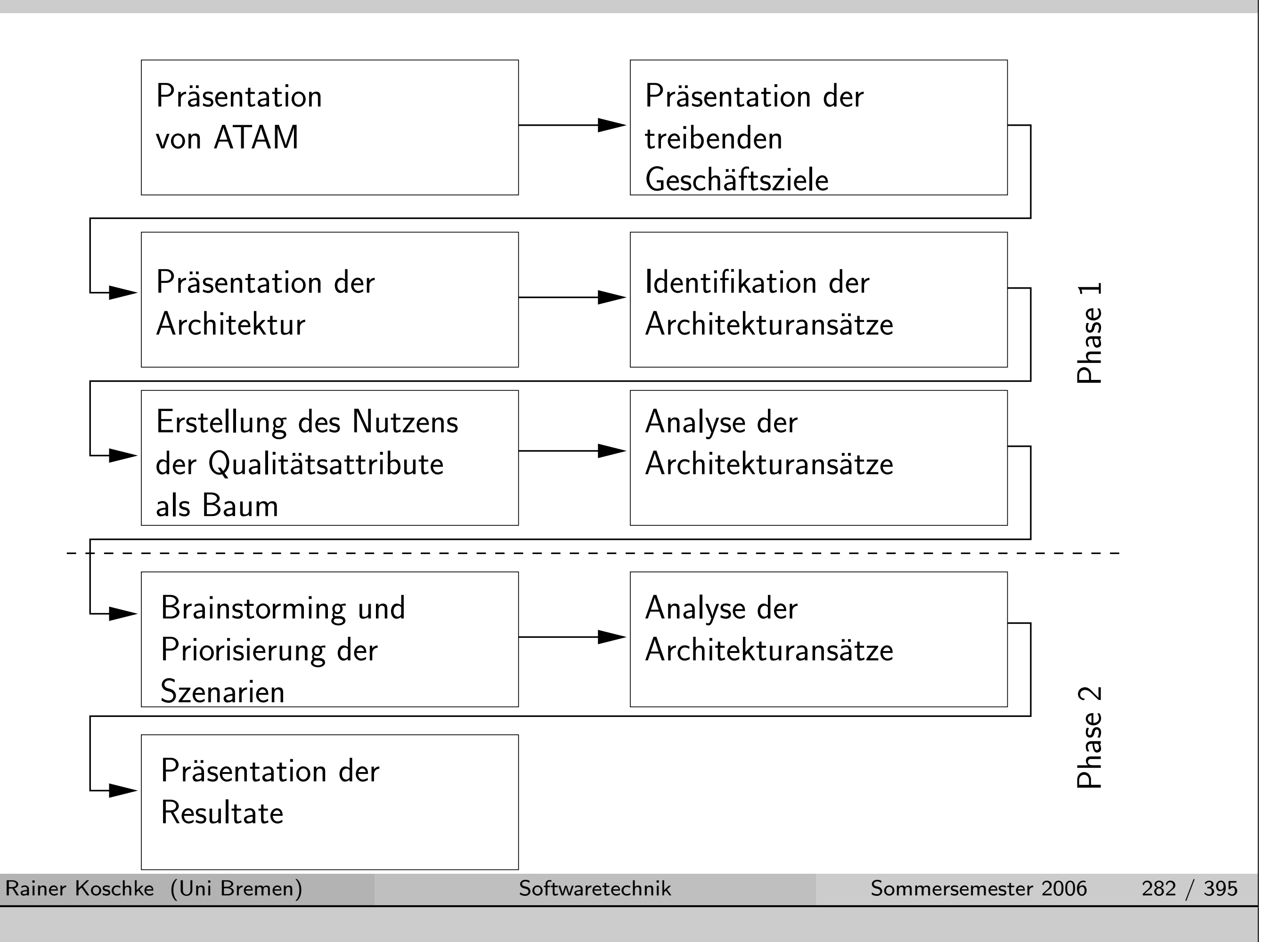
Prozessphasen

Aktivität	Teilnehmer	Dauer
Vorbereitung	Leiter des Evaluationsteams, Schlüsselentscheider	informell über ein paar Wochen
Evaluation Teil I	Evaluationsteam, Architekt und Entscheider	1 Tag gefolgt von einer Pause von 2-3 Wochen
Evaluation Teil II	Evaluationsteam, Architekt, Entscheider, Stakeholders	2 Tage
Wiedervorlage	Evaluationsteam und Betroffene	1 Woche

Prozessphase Vorbereitung

- Einführung ins Projekt
- Erörterung der Logistik
- initiale Liste der Stakeholders
- Zeitplan
- Übergabe verfügbarer Architekturdokumentation

ATAM-Evaluationsphase



Prozessphase Evaluation - Teil I

1. Evaluationsleiter stellt ATAM, Kontext, Erwartungen vor
2. Projektentscheider stellt Geschäftsziele vor
   - die wichtigsten Funktionen des Systems
   - relevante technische, organisatorische, ökonomische oder politische Randbedingungen
   - Geschäftsziele und Kontext
   - wesentliche Stakeholders
   - Hauptqualitätsziele der Architektur

Prozessphase Evaluation - Teil I

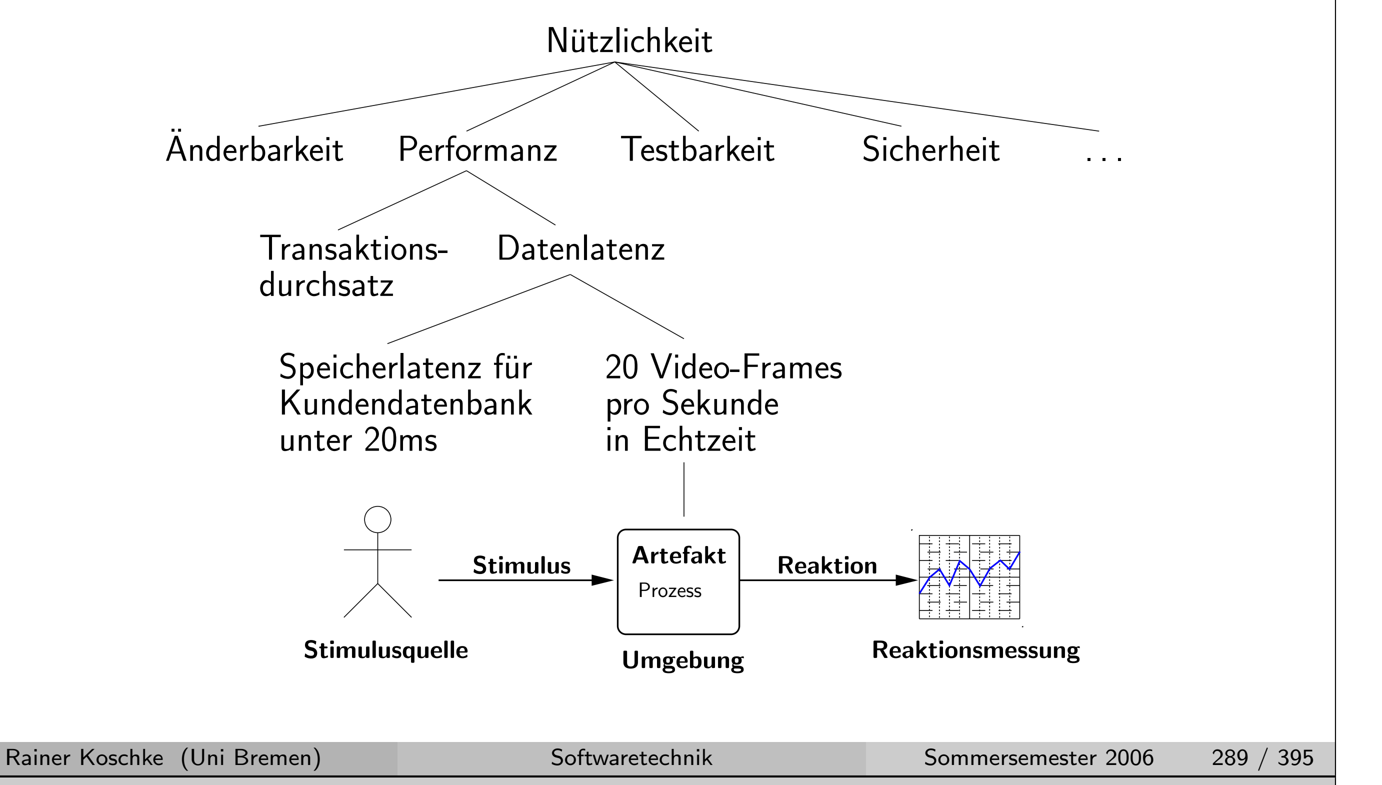
3. Architekt stellt Architektur vor
   - technische Randbedingungen (Betriebssystem, Hardware, Middleware, andere verbundene Systeme etc.)
   - Architekturansätze (Stile und Muster)
   - Struktur der Präsentation:
     - Darstellung der verschiedenen Sichten (z.B. Siemens-Sichten)
     - prinzipielle Entwurfsentscheidungen, Muster, Taktiken
     - Integration von COTS-Komponenten
     - Verfolgung von 1-3 der wichtigsten Anwendungsfälle
     - Verfolgung von 1-3 der wichtigsten Änderungsszenarien
     - Risiken

Prozessphase Evaluation - Teil I

4. Architekturansätze werden identifiziert
   - Muster und Taktiken werden vom Team identifiziert
   - und vom Protokollanten für alle sichtbar festgehalten;
   - erlauben spätere Analyse (bekannte Vor- und Nachteile), z.B.:
     - Schichtenarchitektur positiv für Portierbarkeit, negativ für Performanz
     - Daten-Repository ermöglicht Skalierbarkeit, Abhängigkeiten schwerer zu durchschauen

Prozessphase Evaluation - Teil I

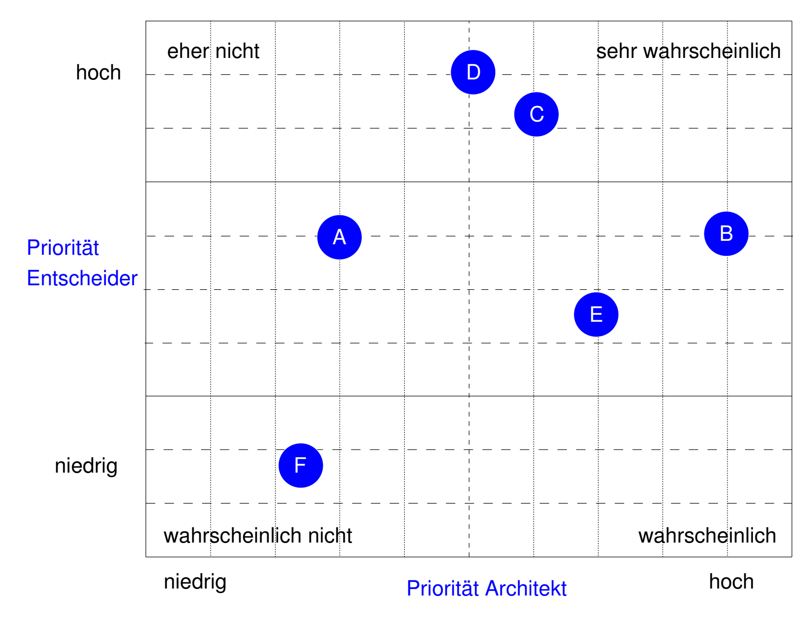
5. Nützlichkeitsbaum für Qualtitätsattribute wird erstellt; Szenarien werden priorisiert



Prozessphase Evaluation - Teil I

Priorisierung der Szenarien

- Priorisierung der Szenarien durch Entscheider
- Priorisierung der Szenarien durch Architektur bezüglich der Schwierigkeit, mit der die Architektur das Szenario erfüllen kann
- Auswahl der Szenarien für Evaluation



Prozessphase Evaluation - Teil I

6. Architekturansätze werden analysiert
   - in Reihenfolge aus Schritt 5 wird jedes Szenario einzeln betrachtet
   - Architekt nimmt Stellung, wie die Architektur das Szenario unterstützt
   - Evaluations-Team (insbesondere Fragesteller) fragt nach den Architekturansätzen
   - Risiken, risikolose Eigenschaften (Nonrisks), Sensitivity Points, Tradeoff Points werden dokumentiert

Beispielanalyse eines Architekturansatzes

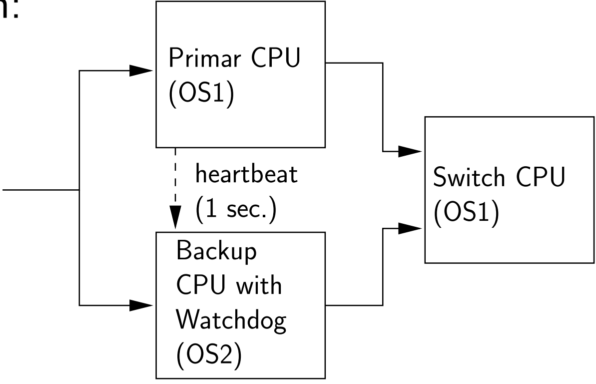
A12	Detect and recover from HW failure of main switch
Attribute(s)	Availability
Environment	Normal Operation
Stimulus	One of the CPUs fails
Response	0.999999 availability of switch

Architectural Decisions	Sensitivity	Tradeoff	Risk	Nonrisk
Backup CPU(s)	S2		R8	
No backup data channel	S3	T3	R9	
Watchdog	S4			N12
Heartbeat	S5			N13
Failover routing	S6			N14

Reasoning:

- Ensures no commom mode failure by using different hardware and operating systems (see Risk 8)
- Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst
- Guaranteed to detect failure within 2 seconds based on rates of heartbeat and watchdog
- Watchdog is simple and provided reliable
- Availability requirement might be at risk due to lack of backup data channel (see Risk 9)





Prozessphase Evaluation - Teil II

7. Brainstorming und Priorisierung der Szenarien
   - Stakeholders schlagen Szenarien vor, die für sie relevant sind

   → falls bis dato nicht berücksichtigt: Uneinigkeit zwischen Architekt und Stakeholder entdeckt → Risiko
   - Szenarien werden priorisiert: jeder Stakeholder kann ein Drittel aller Szenarien als relevant (öffentlich) auswählen (Kumulieren und Panaschieren erlaubt)
   - Selektion der wichtigsten Szenarien (z.B. ab einem deutlichen Sprung der Voten)

Prozessphase Evaluation - Teil II

8. Architekturansätze werden analysiert
   - Architekt erläutert gegenüber Stakeholders, wie Szenarien behandelt werden
   - analog zu Schritt 6

Prozessphase Evaluation - Teil II

9. Präsentation der Resultate
   - Vortrag oder schriftlicher Bericht; Zusammenfassung der Ergebnisse:
     - dokumentierte Architekturansätze
     - priorisierte Szenarien
     - Nützlichkeitsbaum
     - entdeckte Risiken
     - dokumentierte Nonrisks
     - Sensitivity Points und Tradeoff Points
   - Zusammenfassung verwandter Risiken
     - z.B. Architektur beachtet nicht verschiedene Hardware- und Softwareausfälle

     → ungenügende Beachtung von Verfügbarkeit

Prozessphase Wiedervorlage

- Resultat wird vorgestellt/übergeben
- Diskussion über den Verlauf der Evaluation
- Prozessbeobachter berichtet
- Aufwand wird festgehalten

Nach einigen Monaten:

- Langzeiteffekte der Evaluation (sowohl für Architektur als auch weitere Evaluationen) werden bestimmt
- Kosten/Nutzen-Abwägung

Wiederholungs- und Vertiefungsfragen I

- Was ist Software-Architektur?
- Welche Bedeutung kommt ihr zu?
- Was ist eine Architektursicht bzw. -blickwinkel?
- Erläutern Sie die Kategorien von Blickwinkeln von Clements et al.
- Erläutern Sie die Siemens-Blickwinkel. Wozu die Trennung in verschiedene Blickwinkel? Wer hat Interesse an den jeweiligen Blickwinkeln?
- Erläutern Sie den Begriff *Qualität* in Bezug auf Software-Architektur.
- Was ist ein Qualitätsattributsszenario und was bezweckt man damit?
- Was ist eine Taktik im Zusammenhang mit Software-Architektur?
- Nennen Sie Taktiken für Änderbarkeit und Testbarkeit.
- Geben Sie ein Szenario an für das Qualitätsattribut . . . (Performanz, Änderbarkeit, Testbarkeit, aber auch andere).

Wiederholungs- und Vertiefungsfragen II

- Welche grundsätzlichen Techniken gibt es, um Software-Architekturen zu evaluieren?
- Erläutern Sie die *Architecture Tradeoff Analysis Method (ATAM)*.
- Welche Rollen sind bei ATAM vorgesehen?
- Erläutern Sie die Begriffe Sensitivity und Tradeoff Point. Wozu wird der Unterschied gemacht?
- Wann betrachtet man eine Entwurfsentscheidung als Risiko?

Entwurfsmuster

Lernziele

- Verstehen, was Entwurfsmuster sind
- Verschiedene Enwurfsmuster kennen und anwenden können
- Qualitäten und Einsetzbarkeit der Entwurfsmuster kennen

Entwurfsmuster

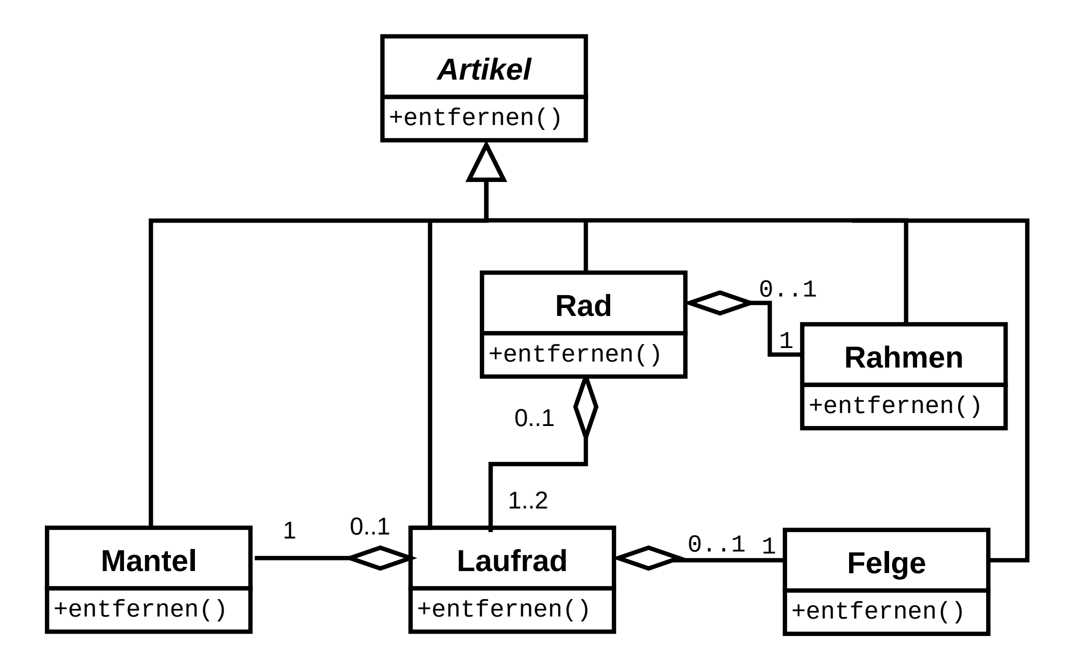
Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander (Architekt und Mathematiker),
"A pattern language", 1977.

Definition

Entwurfsmuster: "Musterlösung" für ein wiederkehrendes Entwurfsproblem.

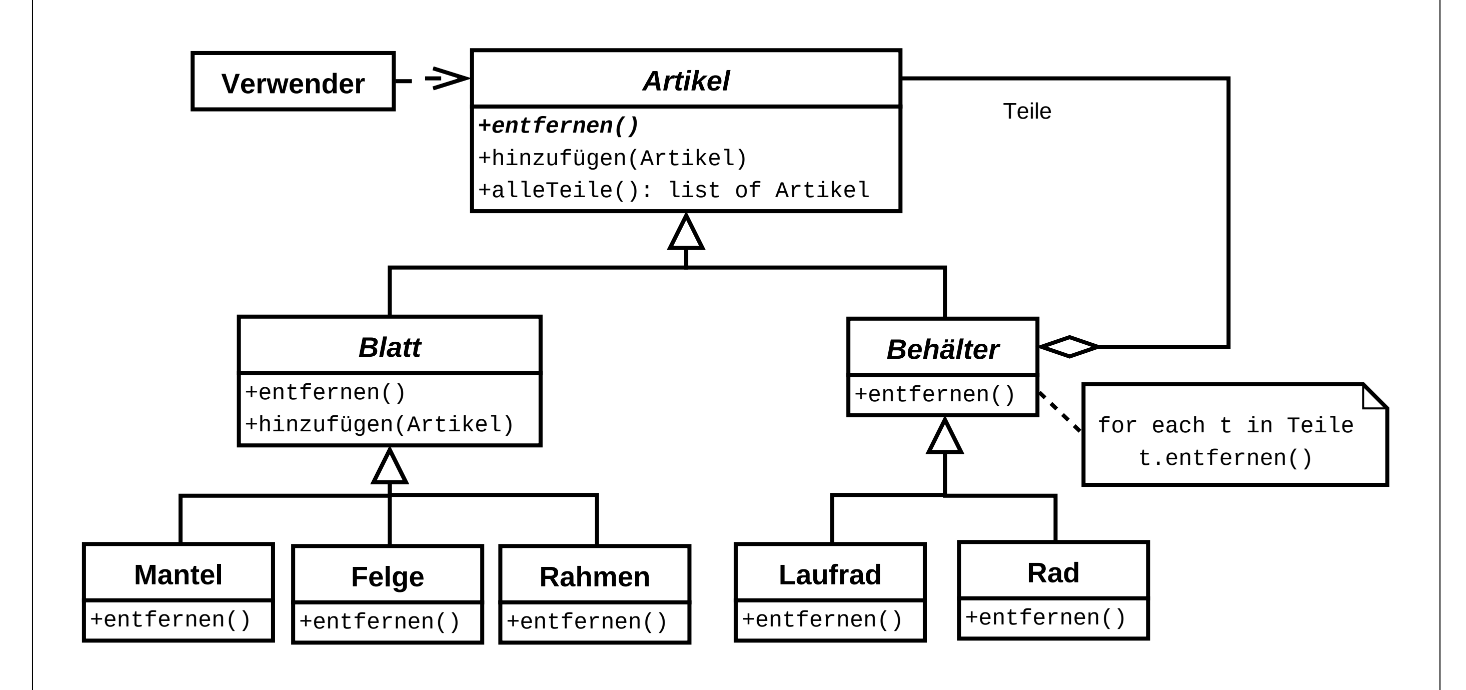
Zusammengesetzte Fahrradelemente



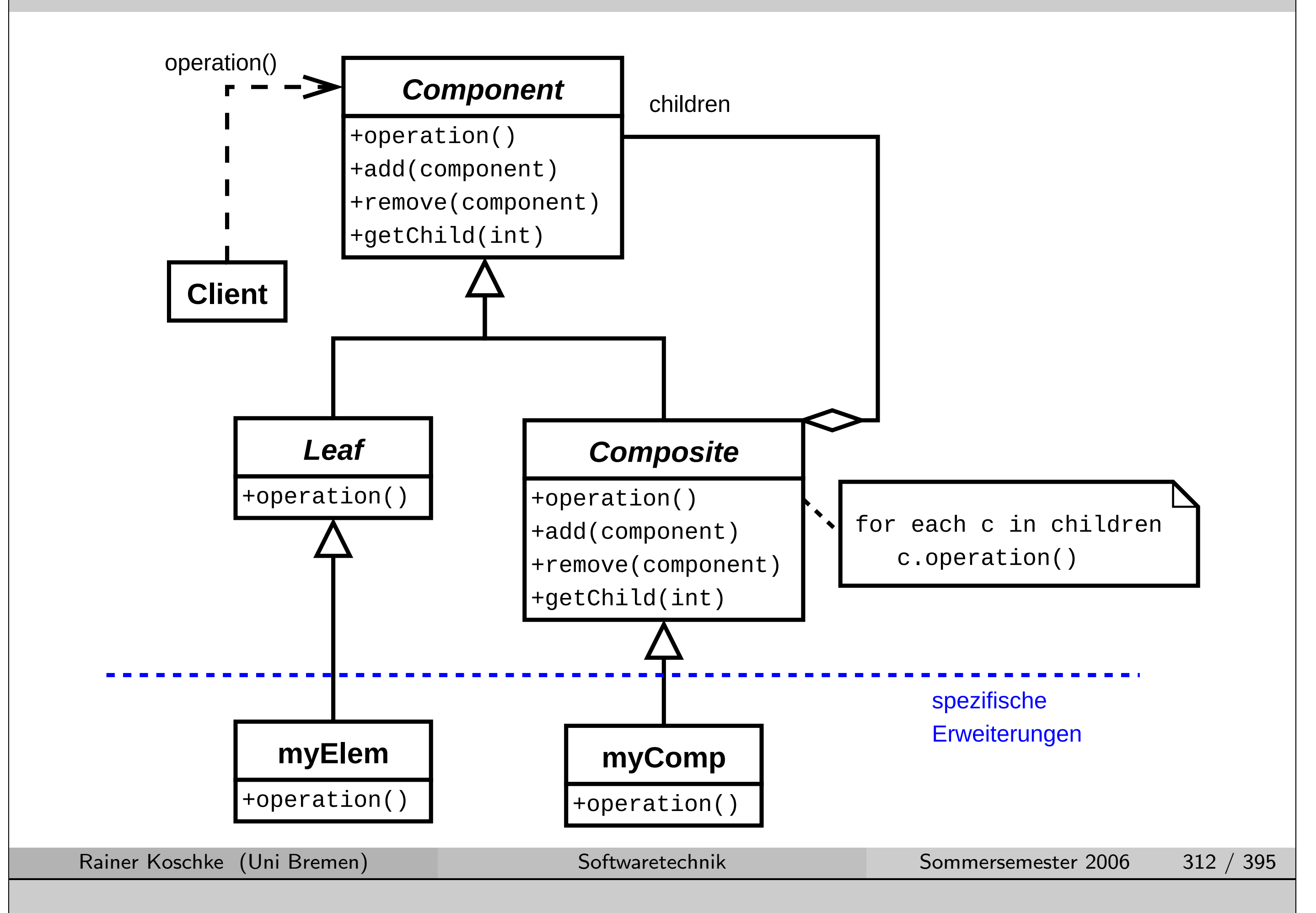
Wir wollen ...

- Teil-von-Hierarchien für Artikel bilden
- Verwender von Artikeln sollen den Unterschied zwischen Kompositionen und Einzelelementen ignorieren können

Modellierung der Teil-von-Hierarchie



Entwurfsmuster *Composite*



Kategorien von Entwurfsmustern

- Muster für das Erzeugen von Instanzen
  - Singleton
- strukturelle Muster zur Komposition von Klassen oder Objekten
  - Composite
  - Adapter
  - Decorator
- Verhaltensmuster betreffen Interaktion von Klassen oder Objekten
  - Command

Bestandteile eines Entwurfsmusters

- **Name** (kurz und beschreibend)
- **Problem**: *Was* das das Entwurfsmuster löst
- **Lösung**: *Wie* es das Problem löst
- **Konsequenzen**: Folgen und Kompromisse des Musters.

Problem

- es soll nur eine einzige Instanz einer Klasse geben, die global verfügbar sein soll
- Beispiele:
  - zentrales Protokoll-Objekt, das Ausgaben in eine Datei schreibt.
  - Druckaufträge, die zu einem Drucker gesendet werden, sollten nur in einen einzigen Puffer geschrieben werden.

Entwurfsmuster Singleton (Einzelstück)

Lösung (Muster für das Erzeugen von Instanzen):

Singleton
- static instance: Singleton - static ...
+ static Singleton getInstance() - Singleton() + ...

Code

```
1  public final class Singleton {
2
3   private static Singleton instance;
4   // speichert einzige Instanz
5
6   private Singleton() {}
7   // kann von außerhalb nicht benutzt werden
8
9   // liefert einzige Instanz
10  public synchronized static Singleton getInstance() {
11    if (instance == null) { // lazy instantiation
12      instance = new Singleton();
13    }
14    return instance;
15  }
16 }
```

Konsequenzen

- Singleton hat strikte Kontrolle über wie und wann Klienten es verwenden
- vermeidet globale Variablen
- leicht erweiterbar, um mehrere Instanzen zuzulassen
- Singleton kann spezialisiert werden

Spezialisierung von Singletons

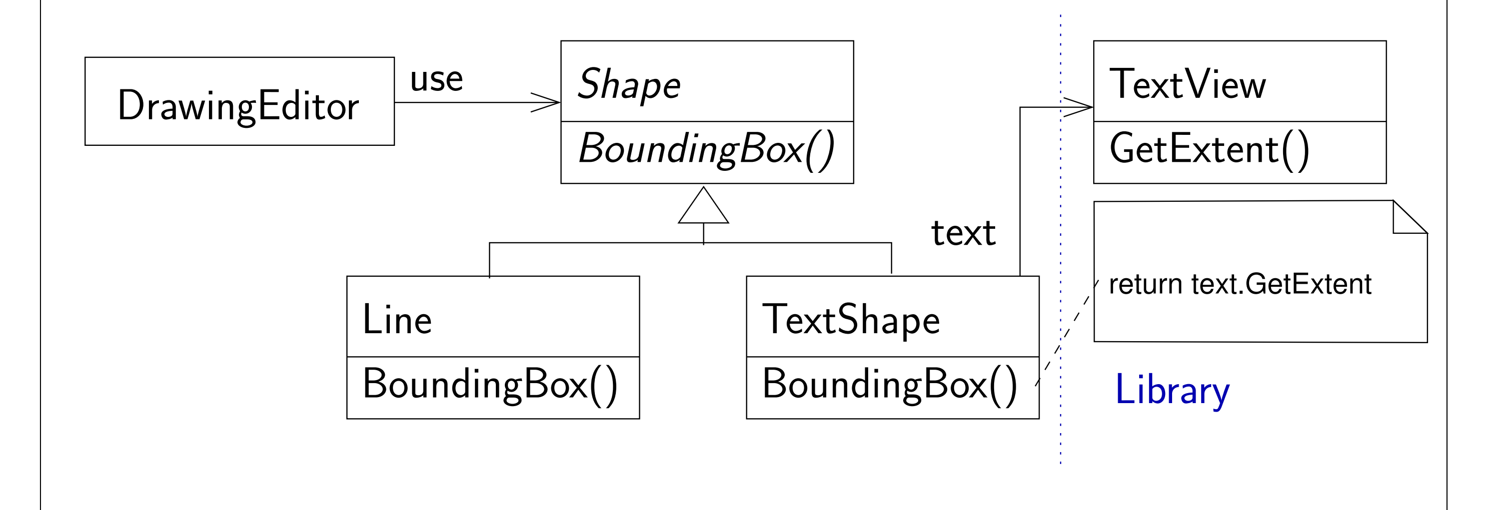
```
1 public class Singleton {
2  protected Singleton() {}
3  // kann von außerhalb nicht benutzt werden
4  private static Hashtable<String, Singleton> registry
5     = new Hashtable<String, Singleton>();
6  // Registry für alle Instanzen von Singleton und Unterklassen
7
8  protected static String ClassKey () {return "Singleton";};
9  // eindeutiger Schlüssel der Klasse;
10 // muss von Unterklasse redefiniert werden
11
12 // liefert einzige Instanz
13 public synchronized static Singleton getInstance()
14  {return registry.get (ClassKey ());};
15
16 // muss vor Benutzung aufgerufen werden;
17 // registriert Instanz; muss von Unterklasse redefiniert werden
18 public static void Init ()
19  {registry.put (ClassKey (), new Singleton());};
20 }
```

Spezialisierung von Singletons

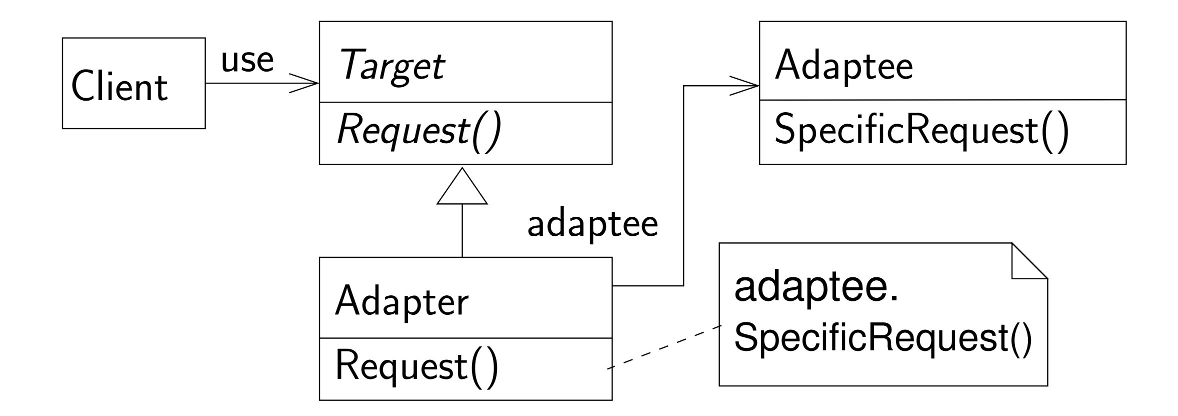
```
1 public class DerivedSingleton extends Singleton {
2
3  protected DerivedSingleton() {};
4
5  protected static String ClassKey ()
6    {return "DerivedSingleton";};
7
8  public static void Init ()
9  {registry.put (ClassKey (), new DerivedSingleton());};
10
11 }
```


Problem

- eine vorhandene wiederverwendbare Komponente hat nicht die passende Schnittstelle,
- und der Code der Komponente kann nicht verändert werden



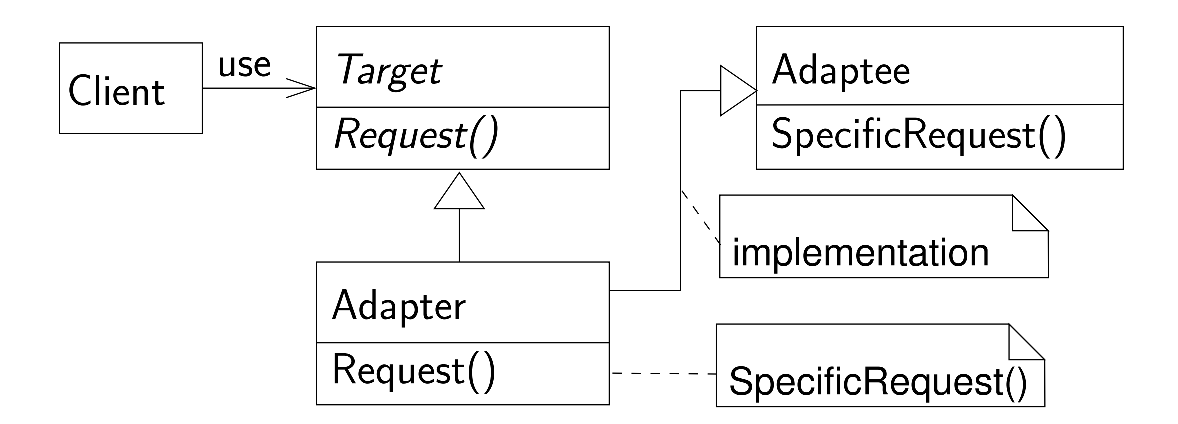
Lösung: Objekt-Adapter



Konsequenzen:

- erlaubt Anpassung von Adaptee und all seine Unterklassen auf einmal
- Overriding von Adaptees Methoden schwierig
  - → Ableitung von Adaptee
  - → Adapter referenziert auf Ableitung
- führt Indirektion ein

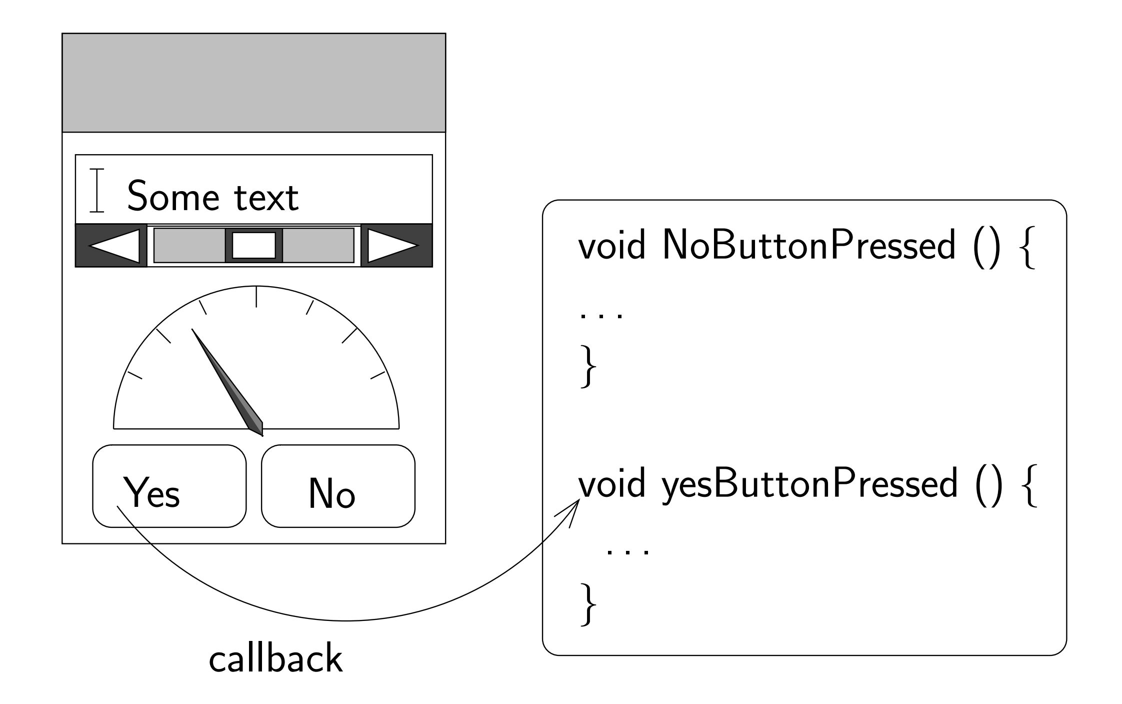
Lösung: Klassen-Adapter



Konsequenzen:

- keine Indirektion
- setzt Mehrfachvererbung voraus
- passt nur Adaptee an, nicht seine Unterklassen
- Overriding von Adaptees Methoden einfach

Problem



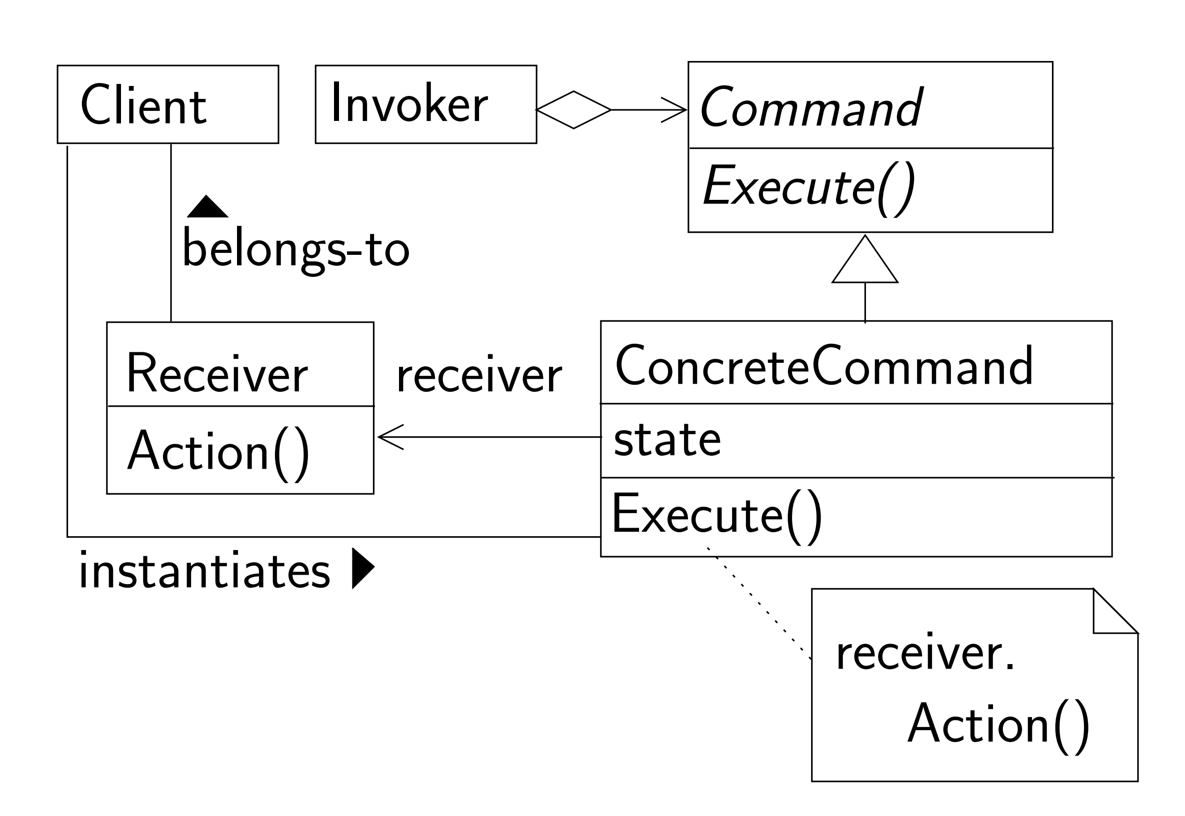
Lösung in C: Klient

```
1 void YesButtonPressed () {...}
2 void NoButtonPressed ()  {...}
3
4 Button mybutton;
5
6 int main (){
7   attach (&mybutton, &YesButtonPressed);
8   ...
9   event_loop ();
10 }
```

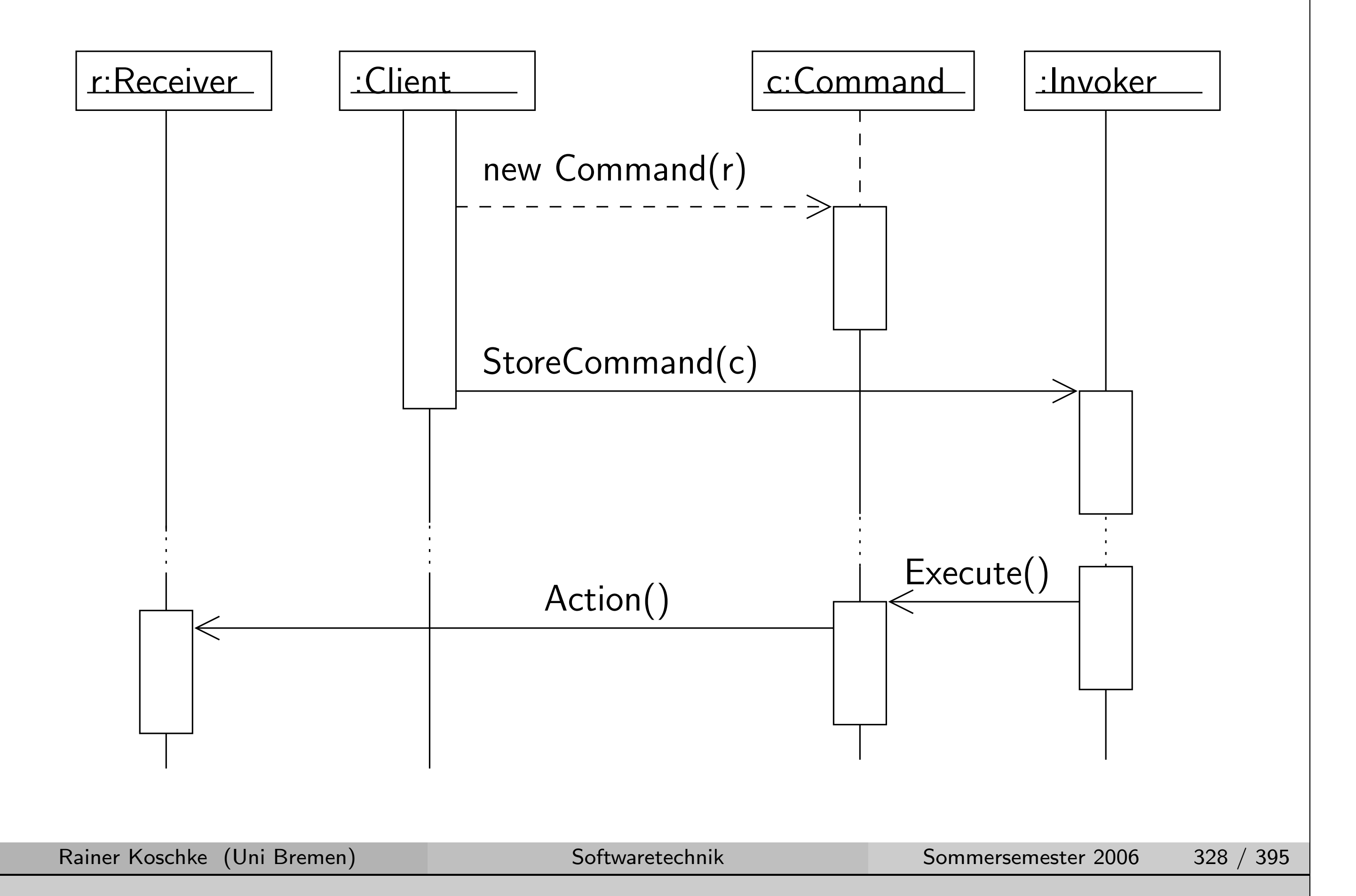
Lösung in C: Mechanismus

```
1  typedef void (*Callback)();
2
3  typedef struct Button {
4    Callback execute;
5    int x;
6    int y;
7  } Button;
8
9  void attach (Button *b, Callback execute) {
10   b->execute = execute;
11 }
12
13 void event_loop (){
14   ...
15   widgets[i]->execute();
16   ...
17 }
```


Lösung mit OOP



Lösung mit OOP

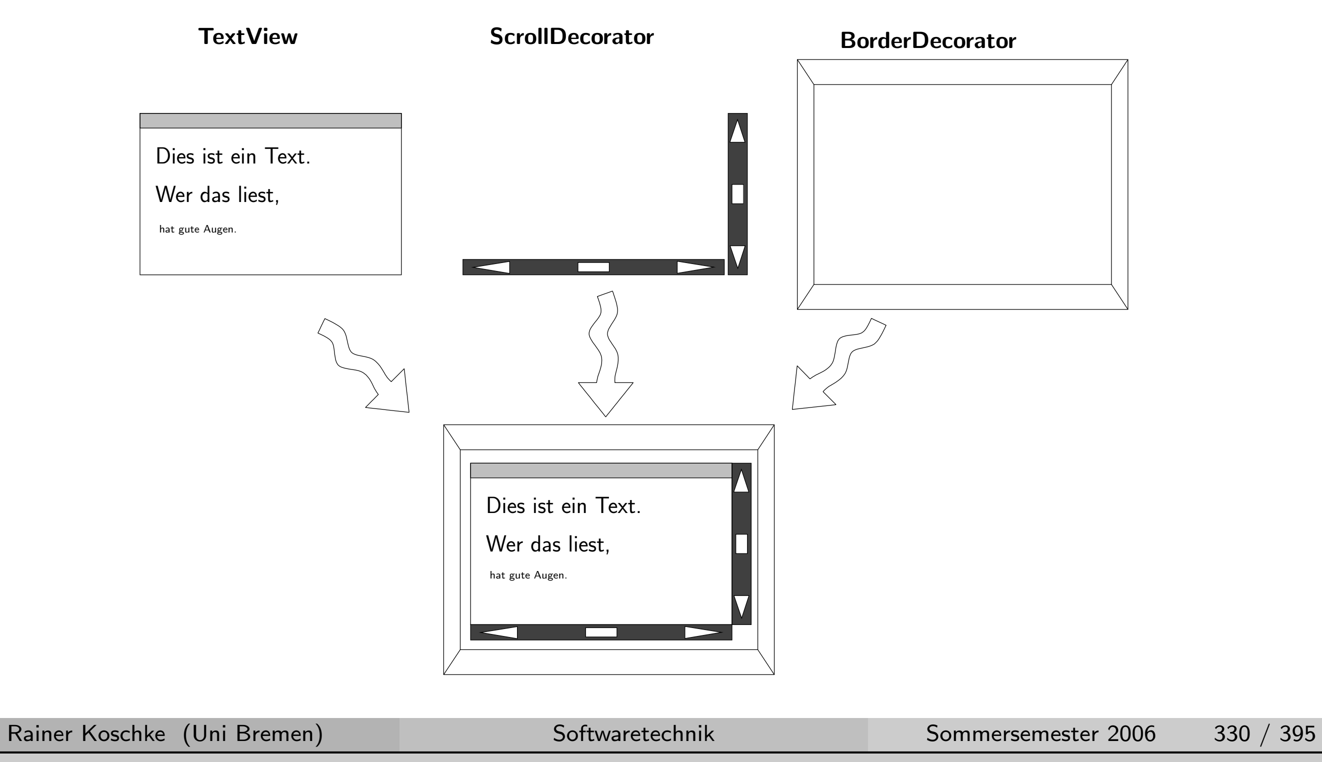


Konsequenzen

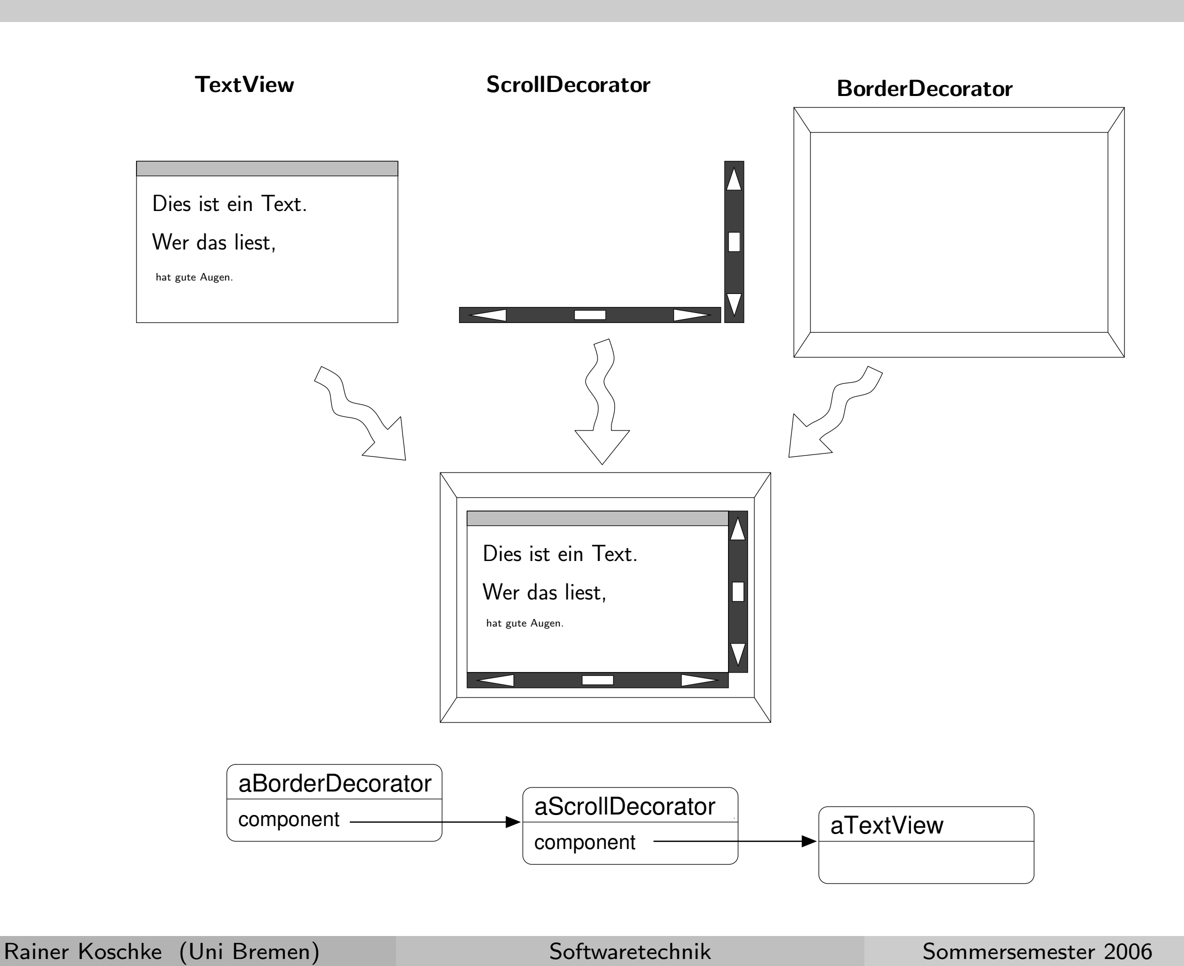
- Entkopplung von Objekt, das Operation aufruft, von dem, welches weiß, wie man es ausführt
- Kommandos sind selbst Objekte und können als solche verwendet werden (Attribute, Vererbung etc.)
- Hinzufügen weiterer Kommandos ist einfach

Problem

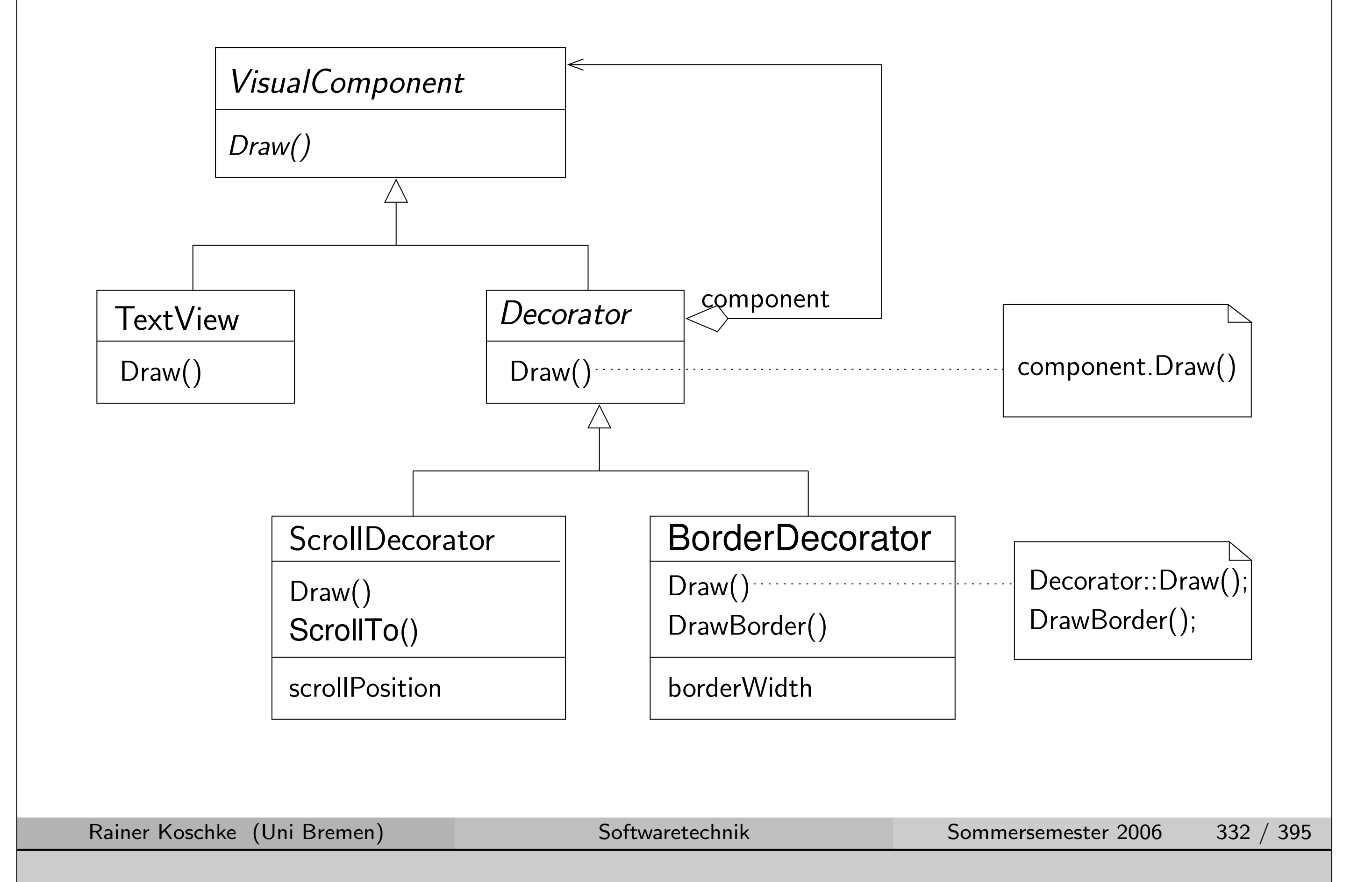
- Eigenschaften sollen einzelnen Objekten, nicht ganzen Klassen, zugewiesen werden können
- Zuweisung soll dynamisch geschehen



Problem



Lösung



Konsequenzen

- höhere Flexibilität als statische Vererbung
- vermeidet Eier-Legende-Wollmilchsau-Klassen
- Dekorator und dekoriertes Objekt sind nicht identisch
- vielfältige dynamische Kombinationsmöglichkeiten → schwer statisch zu analysieren

Weiterführende Literatur

- Das Standardbuch zu Entwurfsmustern ist das von Gamma u. a. (2003)

Wiederholungsfragen

- Was ist ein Entwurfsmuster?
- Warum sind sie interessant für die Software-Entwicklung?
- Wie werden Entwurfsmuster von Gamma et al. kategorisiert?
- Erläutern Sie eines der in der Vorlesung vorgestellten Entwurfsmuster (mit Vor- und Nachteilen und Variationen)[2].

[2] Das *Strategy*-Muster wird bei den Produktlinien vorgestellt und ist prüfungsrelevant. Das *Observer*-Muster wurde im Software-Projekt vorgestellt und ist nicht prüfungsrelevant.

Software-Produktlinien

8 Software-Produktlinien
- Lernziele
- Software-Wiederverwendung
- Erfolgsgeschichten
- Definition
- Übersicht
- Kostenaspekte
- Practice Areas
- Entwicklung der Core-Assets
- Produktentwicklung
- Essentielle Aktivitäten
- Einführung von Produktlinien
- Implementierungsstrategien
- Schwierigkeiten
- Wiederholungsfragen

Lernziele

Software-Produktlinien
- Definition und Bedeutung
- Vor- und Nachteile
- Technische Aspekte
- Organisatorische Aspekte

N.B.: Basiert auf Folien von Linda Northrop
`http://www.sei.cmu.edu/productlines/presentations.html`

Software-Wiederverwendung

1960: Unterprogramme

1970: Module

1980: Objekte

1990: Komponenten

→ opportunistische Wiederverwendung im Kleinen; hat nicht den erwarteten Erfolg gebracht

Software-Produktlinien: geplante Wiederverwendung auf allen Ebene für Familien ähnlicher Systeme

Wiederverwendbares

- Komponenten
- Software-Dokumentation
- Architektur
- Tests (unter anderem Integrations-, Leistungs- und Komponententests)
- Anforderungsspezifikation
- Entwicklungsprozess, Methoden und Werkzeuge
- Budget-/Zeit- und Arbeitspläne
- Handbücher
- Entwickler

Erfolgsgeschichten

CelsiusTech: Familie von 55 Schiffssystemen

- Integrationstest of 1-1,5 Millionen SLOC benötigt 1-2 Leute
- Rehosting auf neue Plattform/Betriebssystem benötigt 3 Monate
- Kosten- und Zeitplan werden eingehalten
- Systemattribute (wie Performanz) können vorausgesagt werden
- hohe Kundenzufriedenheit
- Hardware-/Software-Kostenverhältnis veränderte sich von 35:65 zu 80:20

Erfolgsgeschichten

Nokia: Produktlinie mit 25-30 neuen Produkten pro Jahr
Produktübergreifend gibt es

- unterschiedliche Anzahlen von Tasten
- unterschiedliche Display-Größen
- andere unterschiedliche Produktfunktionen
- 58 verschiedene unterstützte Sprachen
- 130 bediente Länder
- Kompatibilität mit früheren Produkten
- konfigurierbare Produktfunktionen
- Änderung der Geräte nach Auslieferung

Software-Produktlinie

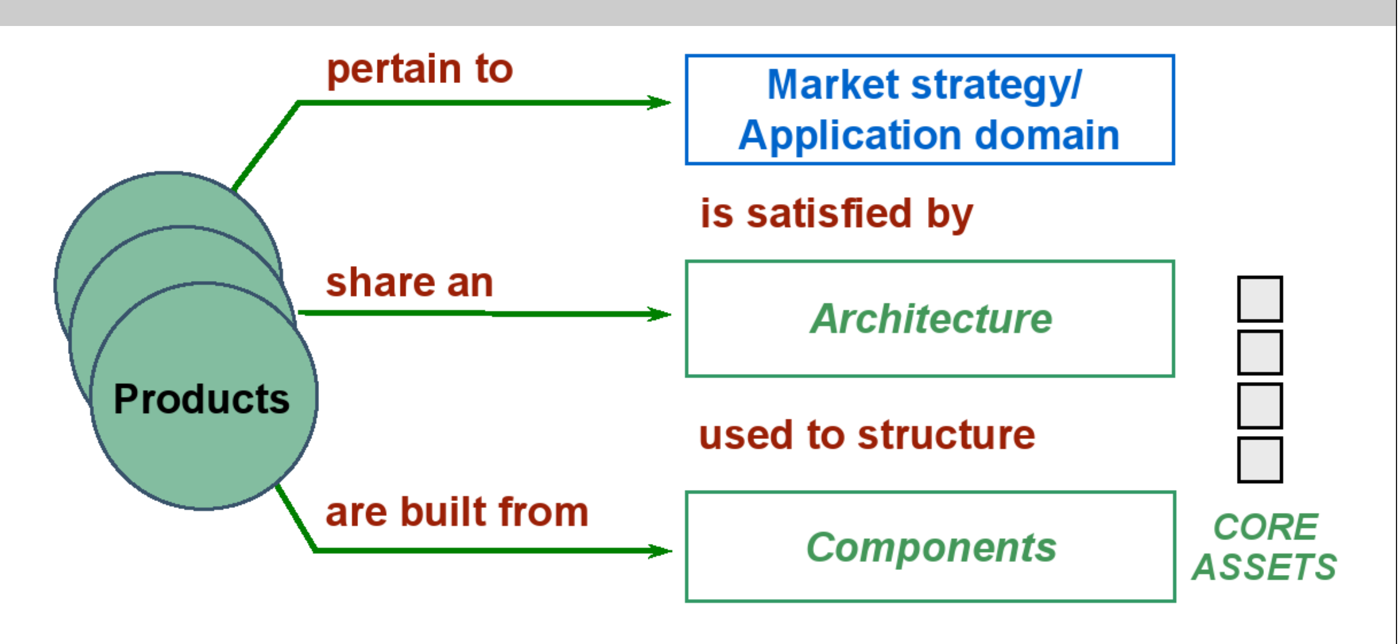
Definition

A **software product line** is a set of software-intensive systems

- sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission
- and that are developed from a common set of core assets in a prescribed way.

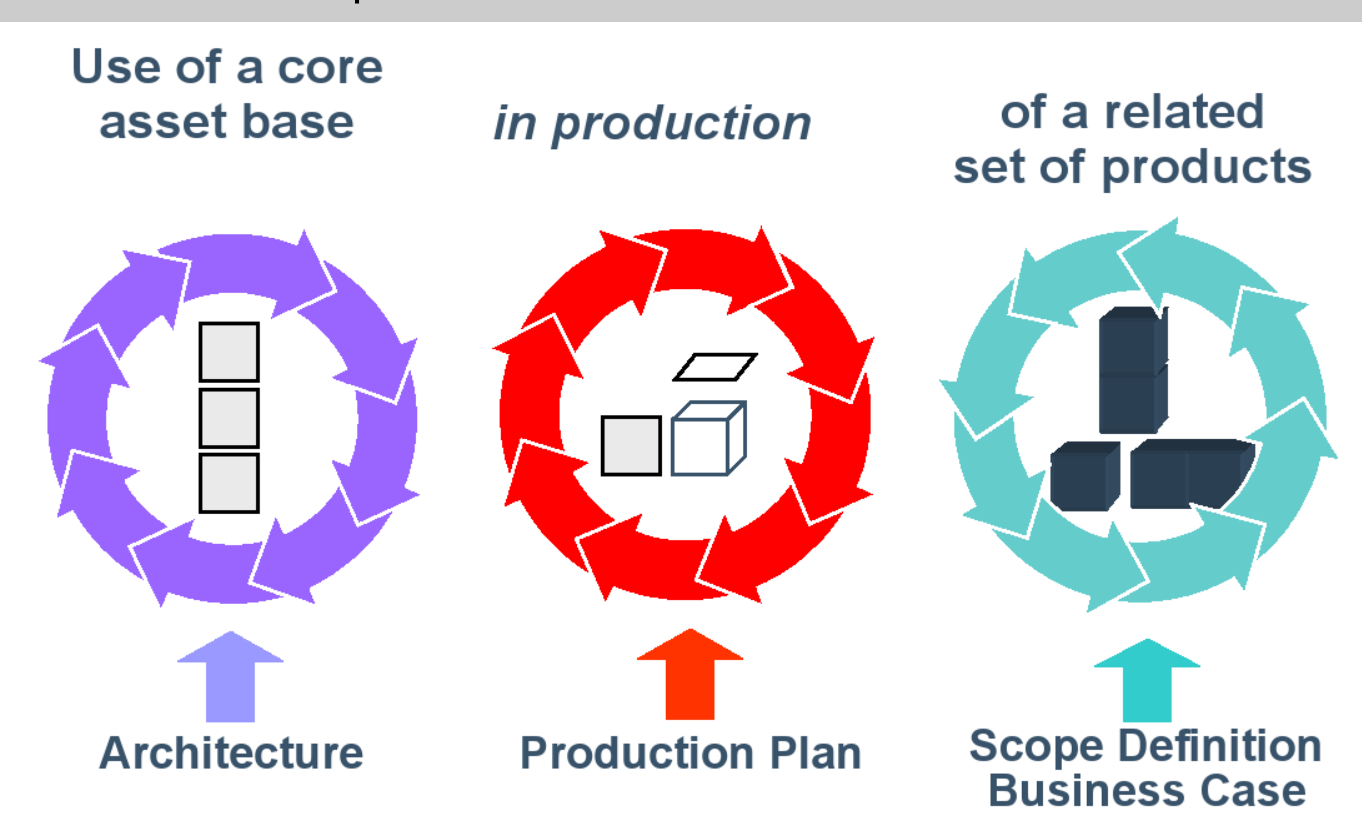
– Clements und Northrop (2001)

Übersicht über Produktlinien



Quelle: Linda Northrop, SEI

Schlüsselkonzepte

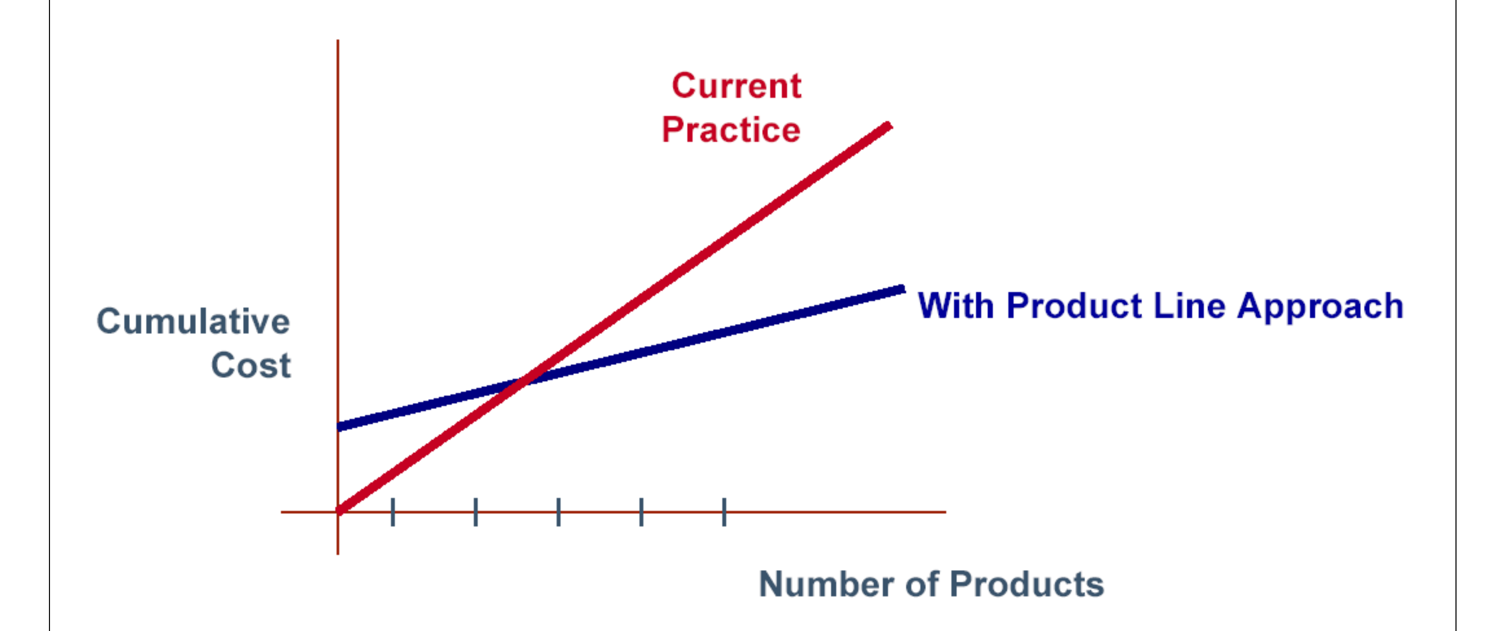


Quelle: Linda Northrop, SEI

Kostenaspekte einer Software-Produktlinie

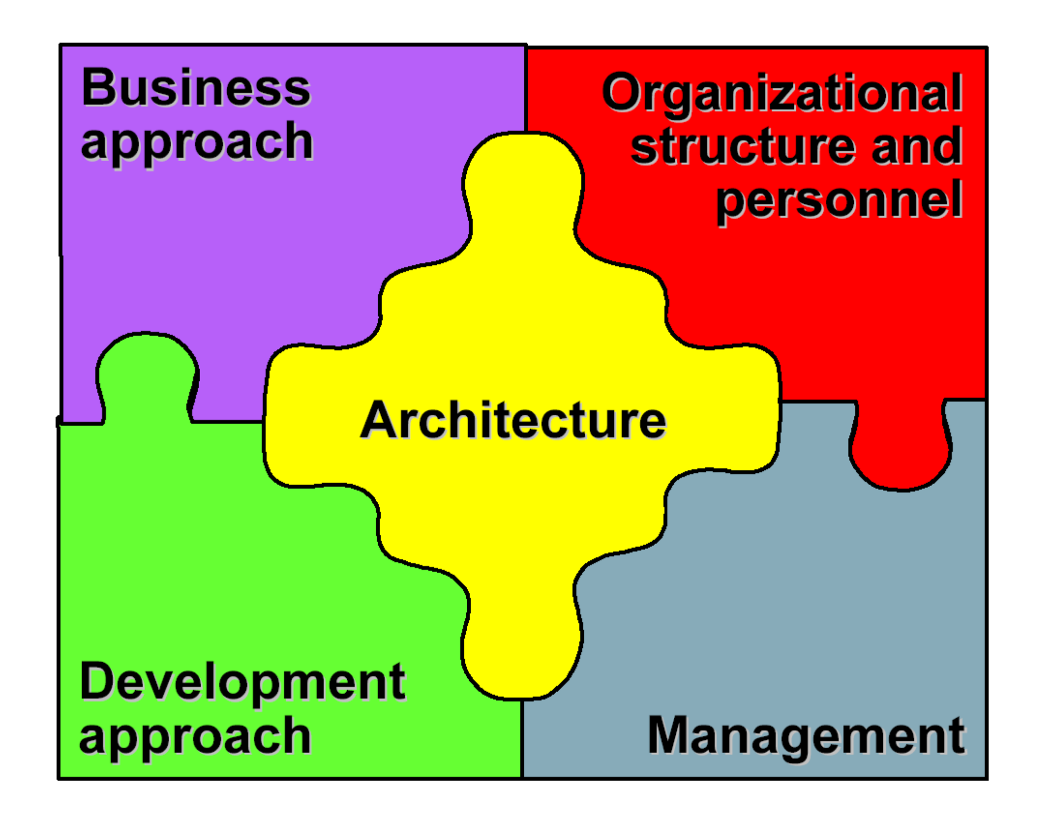
- Marktanalyse: muss eine Familie von Produkten betrachten
- Projektplan: muss generisch oder erweiterbar sein, um Variationen zu erlauben
- Architektur: muss Variation unterstützen
- Software-Komponenten: müssen generischer sein, ohne an Performanz einzubüßen; müssen Variation unterstützen
- Testpläne/-fälle/-daten: müssen Variationen und mehrere Instanzen einer Produktlinie berücksichtigen
- Entwickler: benötigen Training in den Assets und Verfahren der Produktlinie

Return-on-Investment



Quelle: Weiss und Lai, 1999.

Zusammenspielende Komponenten



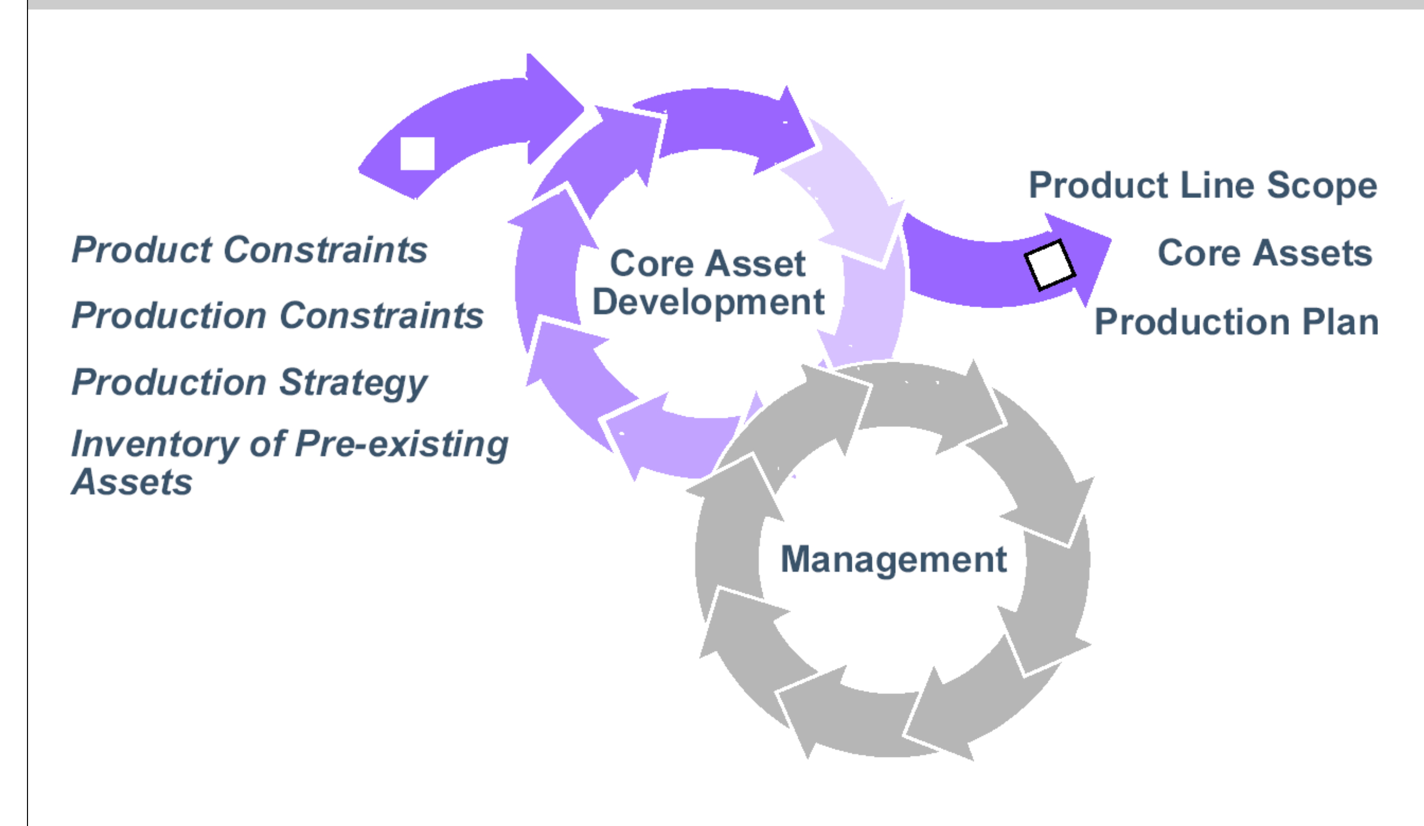
Quelle: Linda Northrop, SEI

Practice Areas



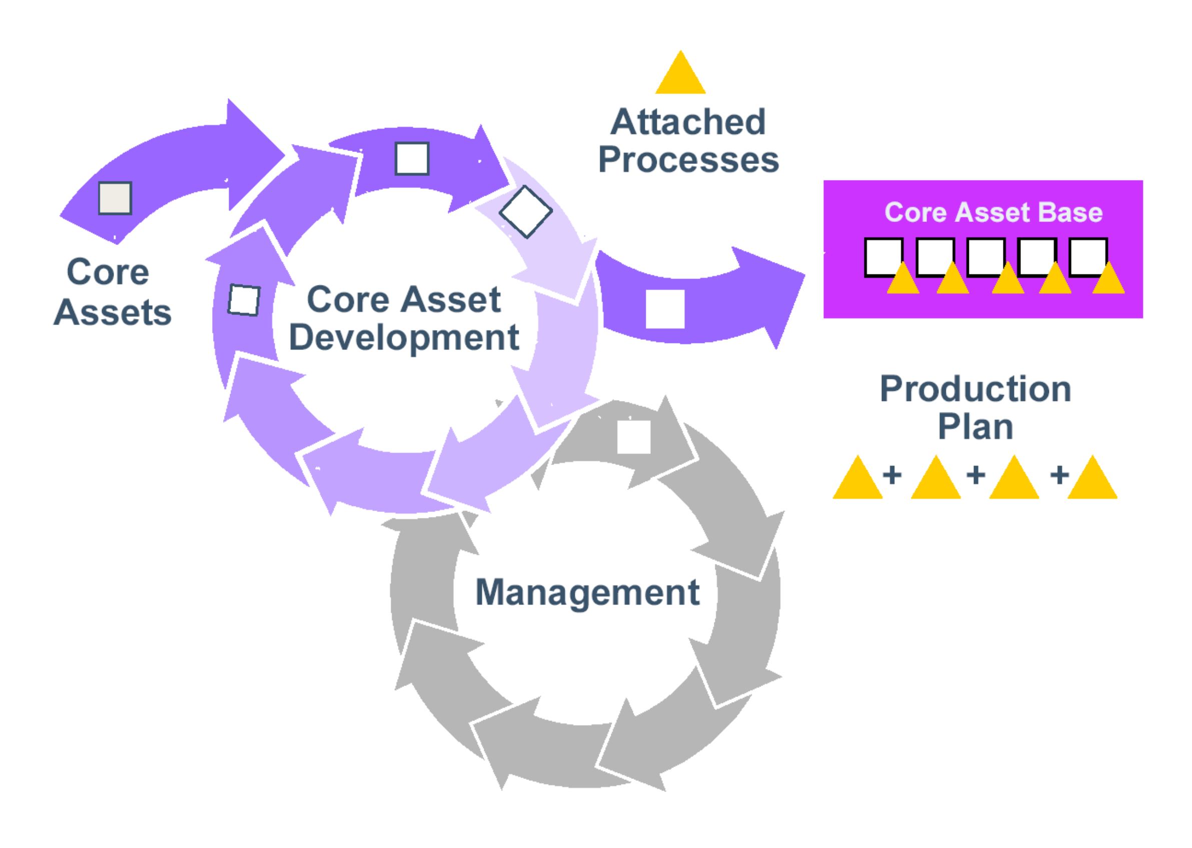
Quelle: Linda Northrop, SEI

Entwicklung der Core-Assets



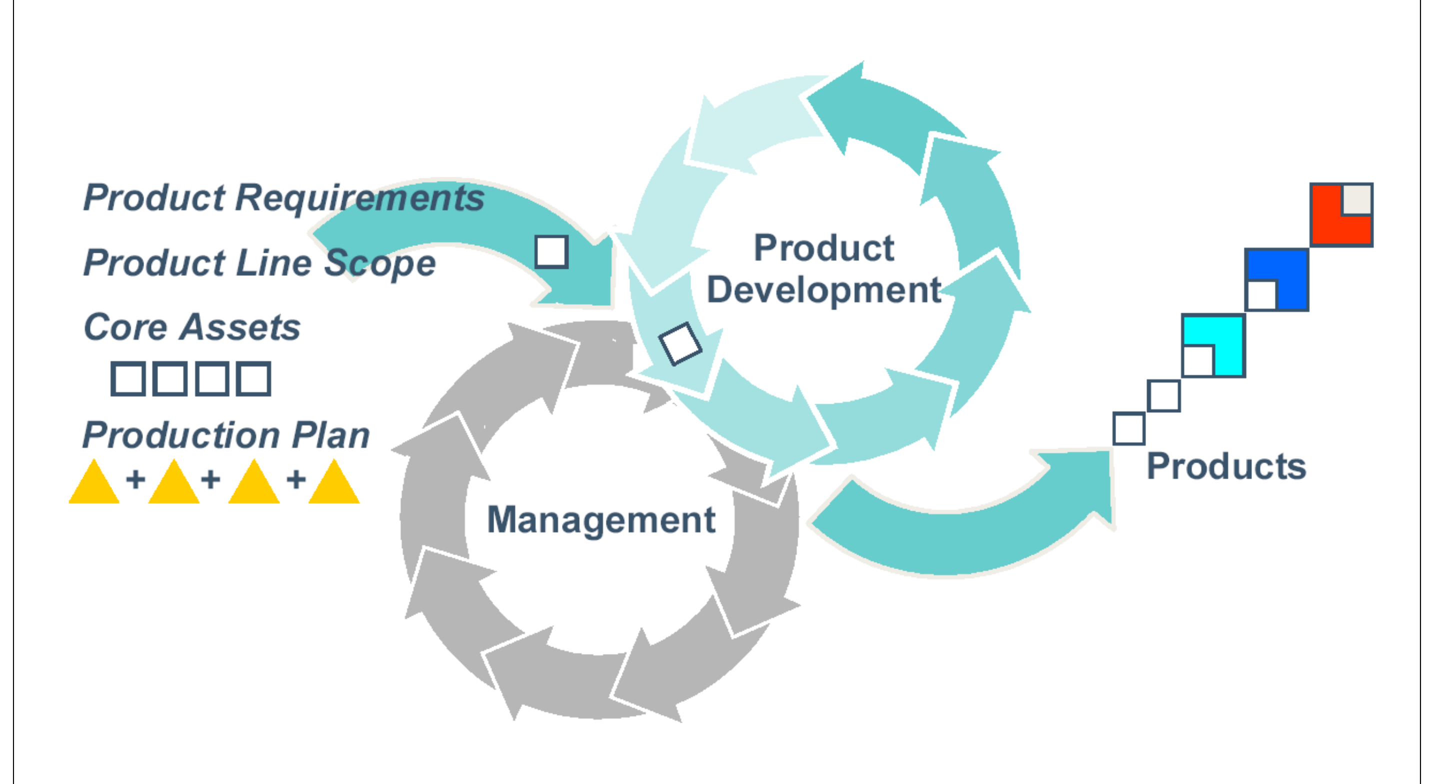
Quelle: Linda Northrop, SEI

Entwicklung der Core-Assets: Assoziierte Prozesse



Quelle: Linda Northrop, SEI

Produktentwicklung



Quelle: Linda Northrop, SEI

Essentielle Aktivitäten



Software Engineering	Technical Management	Organizational Management
Architecture Definition Architecture Evaluation Component Development COTS Utilization Mining Existing Assets Requirements Engineering Software System Integration Testing Understanding Relevant Domains	Configuration Management Data Collection, Metrics, and Tracking Make/Buy/Mine/Commission Analysis Process Definition Scoping Technical Planning Technical Risk Management Tool Support	Building a Business Case Customer Interface Management Implementing an Acquisition Strategy Funding Launching and Institutionalizing Market Analysis Operations Organizational Planning Organizational Risk Management Structuring the Organization Technology Forecasting Training

Practice Areas

Quelle: Linda Northrop, SEI

Einführung von Produktlinien I

Proaktiv:

- definiere zuerst Scope: was gehört zur Produktlinie?
- Scope leitet die weitere Entwicklung
- Entwickle zuerst Core-Assets

+ Produkte können rasch entwickelt werden, sobald die Produktlinie steht

- hohe Vorausleistung und Vorhersagefähigkeit verlangt

Einführung von Produktlinien II

Reaktiv:

- Beginne mit einem oder mehreren Produkten
- Extrahiere daraus Core-Assets für die Produktlinie
- Scope entwickelt sich dabei stetig

+ niedrige Einstiegskosten
+ größerer Einfluss von Erfahrung

- Architektur könnte suboptimal sein, wird schrittweise weiterentwickelt
- Restrukturierungsaufwand notwendig

Einführung von Produktlinien III

Inkrementell (sowohl bei reaktiver als auch proaktiver Entwicklung möglich):

- schrittweise Entwicklung der Core-Assets mit initialer Planung der Produktlinie:
- entwickle Teile der Core-Asset-Base einschließlich Architektur und Komponenten
- entwickle ein oder mehrere Produkte
- entwickle weitere Core-Assets
- entwickle weitere Produkte
- entwickle Core-Asset-Base weiter
- ...

Bindung

Produktlinien ...

- haben Gemeinsamkeiten
- und definierte Unterschiede: Variabilitäten

Produkt wird aus Core-Assets zusammengebaut.
Variabilitäten werden festgelegt.

Bindungszeitpunkt der Variabilitäten

- zur Übersetzungszeit
- zur Bindezeit
- zur Laufzeit

Architekturmechanismen für Variabilitäten

Kombination, Ersetzung und Auslass von Komponenten (auch zur Laufzeit)

Frontend {C, C++, Java}	Middle End {ME}	Backend {i386, Motorola 68000}
C	ME	i386
C	ME	Motorola 68000
C++	ME	i386
C++	ME	Motorola 68000
Java	ME	i386
Java	ME	Motorola 68000

Architekturmechanismen für Variabilitäten

Parametrisierung (einschließlich Makros und Templates)

```
1 generic
2   type My_Type is private;
3   with procedure Foo (M : My_Type);
4 procedure Apply;
5
6 procedure Apply is
7   X : My_Type;
8 begin
9
10     ...
11     Foo (X);
12     ...
13 end Apply;
```


Architekturmechanismen für Variabilitäten

Parametrisierung (einschließlich Makros und Templates)

```
1 typedef (*FP)(int);
2 void Apply (FP fp) {
3     ...
4     fp (X);
5     ...
6 }
```

Architekturmechanismen für Variabilitäten

Selektion verschiedener Implementierung zur Übersetzungszeit (z.B. #ifdef oder Makefile)

```
1 #ifdef Kunde1
2 #define My_Type int
3 void Foo (My_Type M) {...}
4 #else
5 typedef struct mystruct {...} mystruct;
6 #define My_Type mystruct
7 void Foo (My_Type M) {...}
8 #endif
9 void Apply () {
10  My_Type X;
11     ...
12     Foo (X);
13     ...
14 }
```


Architekturmechanismen für Variabilitäten

OO-Techniken: Vererbung, Spezialisierung und Delegation (Entwurfsmuster)

```
1 typedef enum Strategy {s1, s2, s3} Strategy;
2 void Apply (Strategy s) {
3   switch (s) {
4     case s1:  ApplyS1(); break;
5     case s2:  ApplyS2(); break;
6     case s3:  ApplyS3(); break;
7   }
8 }
```

Architekturmechanismen für Variabilitäten

OO-Techniken: Vererbung, Spezialisierung und Delegation (Entwurfsmuster)

```
1 class Applier {
2  Strategy *_strategy
3
4  void Apply () {
5     _strategy->Algorithm ();
6  }
7 }
8 class Strategy {
9    virtual void Algorithm () = 0;
10 }
11 class Strategy1 : Strategy {
12    virtual void Algorithm () {...}
13 }
14 ...
```


Architekturmechanismen für Variabilitäten

Generierung (z.B. Yacc: Grammatik → Code) und aspektorientierte Programmierung

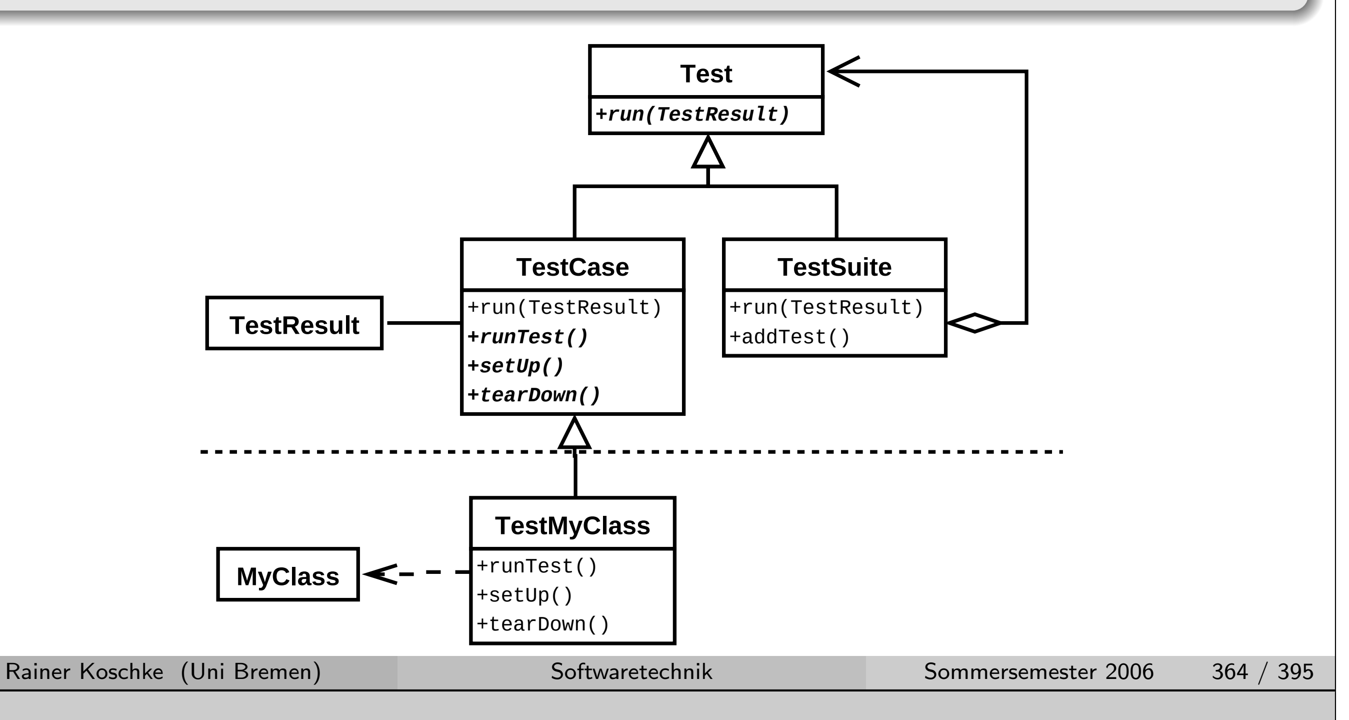
```
1 aspect SetsInRotateCounting {
2     int rotateCount = 0;
3
4     before(): call(void Line.rotate(double)) {
5         rotateCount++;
6     }
7 }
```

- Wie oft wird Methode Line.rotate aufgerufen?

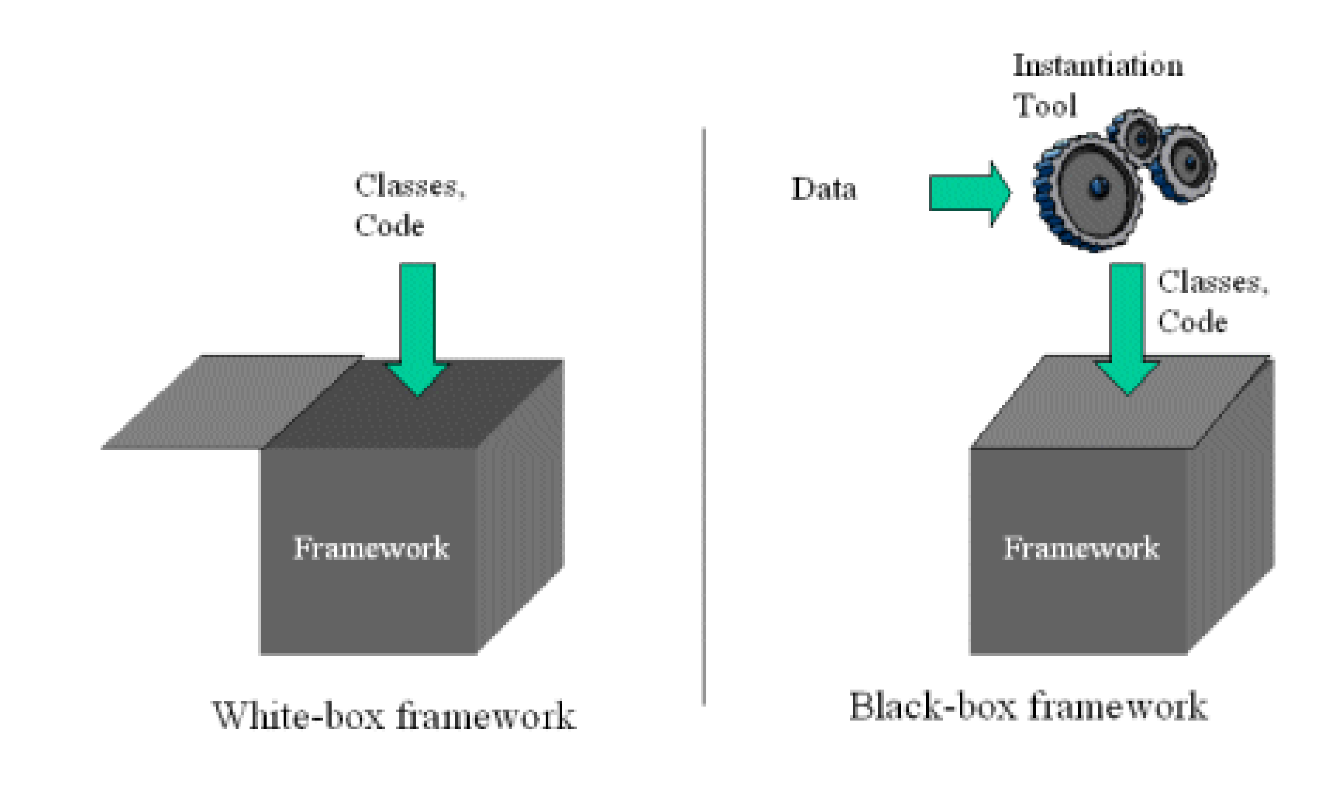
Architekturmechanismen für Variabilitäten

Definition

Anwendungsrahmenwerke (Frameworks): A framework is a set of classes that embodies and abstract design for solutions to a family of related problems. Johnson und Foote (1988)



Anwendungsrahmenwerke (Frameworks)



Schwierigkeiten bei Produktlinien

- Änderung der Organisation (Kern-/Produktaufteilung)
- Was gehört zum Kern?
- Änderungen im Kern haben Auswirkungen auf alle Produkte
- Viele Probleme sind noch nicht gelöst:
  - Test
  - Konfigurationsmanagement
  - ...

Wiederholungs- und Vertiefungsfragen

- Erläutern Sie die Ideen von Software-Produktlinien. Was verspricht man sich davon?
- Was sind die Vor- und Nachteile?
- Wie wird die Entwicklung von Software-Produktlinien organisatorisch häufig strukturiert?
- Erläutern Sie einige essentielle Aktivitäten und ihre Besonderheiten im Kontext von Software-Produktlinien.
- In welcher Weise können Produktlinien eingeführt werden?
- Beschreiben Sie Implementierungsmechanismen für die Variabilität in Software-Produktlinien. Nennen Sie hierbei den jeweiligen Bindungszeitpunkt (was drückt der Bindungszeitpunkt aus?).

Empirische Softwaretechnik

9. Empirische Softwaretechnik
   - Motivation
   - Wissenserwerb in Wissenschaft und Engineering
   - Untersuchungsmethoden
   - Bestandteile eines Experiments
   - Wiederholungsfragen

Lernziele

- die Notwendigkeit zur empirischen Forschung in der Softwaretechnik erkennen
- prinzipielles Vorgehen verstehen
- (irgendwann einmal) empirisch forschen können

Experimente in der Softwaretechnik

Experimentation in software engineering is necessary but difficult. Common wisdom, intuition, speculation, and proofs of concept are not reliable sources of credible knowledge.

– V.R. Basili, 1999

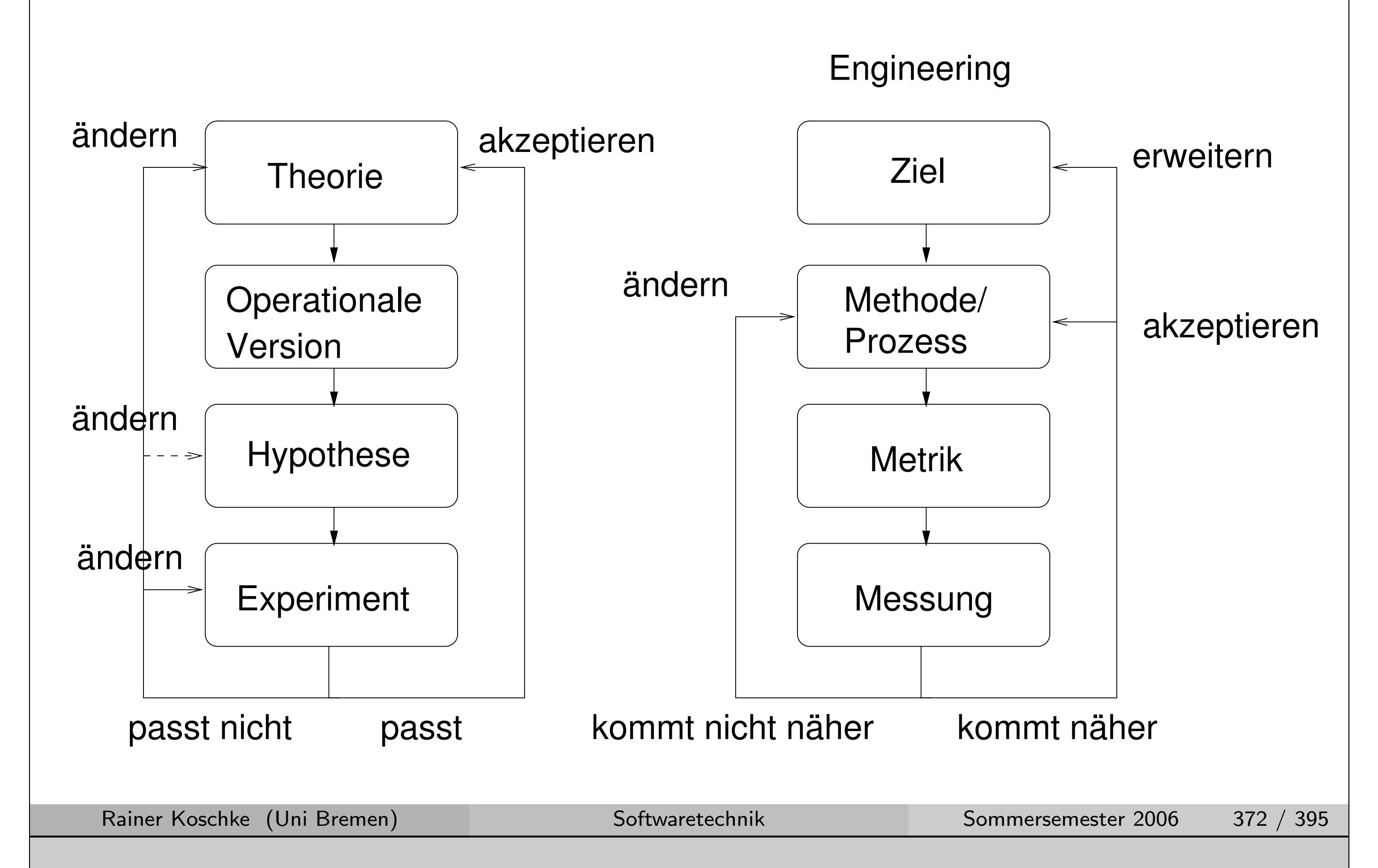
Motivation

- Wir wollen genau wissen, ob und unter welchen Randbedingungen eine Methode funktioniert.
- Forschung
  - beweist durch logische Schlüsse
  - oder aber beobachtet, experimentiert und misst.
- Messung ist sorgfältige Beobachtung mit größtmöglicher Präzision, Zuverlässigkeit und Objektivität
- Messungen identifizieren neue Phänomene, testen Hypothesen oder leiten uns bei der Anwendung von Modellen und Methoden

Empirische Untersuchungen.

Methoden, die von Menschen angewandt werden, können nur empirisch untersucht werden.

Wissenserwerb in Wissenschaft und Engineering



Untersuchungsmethoden

Umfragen und Erhebungen:

- Datenerfassung mit Fragebögen oder ethnographische Studien
- können auch a-posteriori durchgeführt werden
- dienen oft der Bildung von Hypothesen für nachfolgende Fallstudien oder kontrollierte Experimente
- fundierte Methoden zur Datenerhebung und -validierung notwendig
- entstammen den Sozialwissenschaften und kognitiven Wissenschaften

Untersuchungsmethoden

Fallstudien (Quasi-Experimente):

- In-Vivo-Experiment oder Feldstudien mit Hypothese
- eingebettet in echte Projekte und damit weniger kontrolliert
- Herausforderung: zu messen, ohne den Verlauf zu verfälschen

Untersuchungsmethoden

kontrollierte Experimente (In-Vitro-Experiment):

- finden in kontrollierter Testumgebung statt
- Hypothese wird falsifiziert oder bestätigt (mit einer gewissen Wahrscheinlichkeit)
- unabhängige Variablen: Eingabeparameter, die im Experiment variiert werden
- abhängige Variablen: Ausgabeparameter, die gemessen werden

Bestandteile eines Experiments

1. Festlegung der Ziele

Aspekte:

1. Objekt der Studie (z.B. Entwurf, Codierung, Test)
2. Zweck der Studie (z.B. Vergleich, Analyse, Voraussage)
3. Fokus der Studie (z.B. Effektivität, Effizienz)
4. Standpunkt (z.B. Praktiker, Forscher)
5. Kontext (z.B. Erfahrung der Teilnehmer, verwendete Elemente, Umgebung)

Kontext → unabhängige Variablen
Fokus → abhängige Variablen

Bestandteile eines Experiments

2. Formulierung einer testbaren Hypothese

„Methode M ist geeignet für Aufgabe A“
versus
„Methode M benötigt weniger Zeit als Methode N, um Aufgabe A zu erledigen“

Null-Hypothese (Negation der Hypothese):
„Es gibt keinen Unterschied zwischen M und N, um Aufgabe A zu erledigen“

Wenn Null-Hypothese widerlegt ist, wird Hypothese bestätigt (mit einem gewissen Grad an Konfidenz)

Bestandteile eines Experiments

3. Aufbau des Experiments

- Korrelation versus Kausalität
- Validität
  - interne: es werden tatsächlich nur die Beziehungen zwischen unabhängigen und abhängigen Variablen gemessen (z.B. Lerneffekte, zeitliche Aspekte, Auswahl der Teilnehmer)
  - externe: Übertragbarkeit (z.B. Studenten versus Praktiker)
- Identifikation aller unabhängiger und abhängiger Variablen
- Randomisierung

→ viele Bücher zu Standard-Designs abhängig von den Rahmenbedingungen

Bestandteile eines Experiments

4. Analyse und Validierung der Resultate

- Auswertung durch statistische Methoden (Korrelationen und Regressionsanalyse)
- Problem des geringen Datenumfangs
  - parametrische statistische Tests setzen bestimmte Verteilung voraus
  - meist nur nicht-parametrische statistische Tests adäquat, weil keine Verteilung voraus gesetzt wird (insbesondere für Nominal- bis Ordinalskalen)
- quantitative Analyse oft ergänzt durch qualitative
- Validierung
  - Befragungen der Teilnehmer, um sicher zu stellen, dass sie alle Fragen verstanden haben
  - Untersuchung statistischer Ausreißer
- Benchmarking schwierig, weil Firmen ihre Daten ungern veröffentlichen

Bestandteile eines Experiments

5. Replikation

- Wiederholung, um „Glückstreffer“ auszuschließen
- Experiment muss detailliert beschrieben sein
- bedauerlicherweise schwer zu veröffentlichen, weil keine neuen Ergebnisse präsentiert werden

Bestandteile eines Experiments

6. Ethische Fragen

- keine Teilnahme ohne explizite Einwilligung
- kein Missbrauch
- Anonymität muss gewährt werden (doppelt blind)
- kein materieller oder sonstiger Gewinn (Bezahlung, Gehaltserhöhung, gute Note etc.)

Bestandteile eines Experiments

7. Grenzen der experimentellen Forschung

- hoher Aufwand (allerdings: Lernen ohne Experimente ist auch teuer)
- Abhängigkeit von menschlichen Versuchsteilnehmern
  - Studenten haben möglicherweise nicht die Erfahrung
  - Praktiker haben wenig Zeit
- Transferierbarkeit der Resultate
  - Software-Projekte haben eine Unzahl möglicher Variablen
  - nicht alle sind kontrollierbar
  - und wenn sie kontrolliert sind, treffen sie möglicherweise nicht auf andere Umgebungen zu

Bestandteile eines Experiments

8. Beschaffenheit empirischer Forschung

- in der Softwaretechnik erst seit ungefähr 1980 als wesentliche Disziplin anerkannt
- heute: Konferenzen und Zeitschriften
- Verschiebung weg von rein mathematischen Methoden
- Mehrzahl der Probleme der Softwaretechnik sind nicht mathematischer Art, hängen vielmehr von Menschen ab

Weiterführende Literatur

- Endres und Rombach (2003) beschreiben wesentliche empirische Kenntnisse in der Software-Technik und brechen eine Lanze für die empirische Forschung in diesem Gebiet.
- Winner u. a. (1991) beschreiben experimentelle Designs und ihre statistischen (parametrischen) Auswertungen
- Lienert (1973) beschreibt verteilungsfreie (nicht-parametrische) statistische Tests

Wiederholungsfragen

- Welche Arten der Untersuchungsmethoden stehen zur Verfügung?
- Nennen Sie deren Vor- und Nachteile.
- Wozu muss man sich bei kontrollierten Experimenten jeweils Gedanken machen?
- Sie sollen Methode M bzw. Werkzeug W bewerten. Wie gehen Sie vor?

1 **Albrecht 1979** Albrecht, Alan: Measuring application development productivity. Monterey, CA, USA, 1979

2 **Balzert 1997** Balzert, Helmut: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 1997. – ISBN 3827400651

3 **Banker u. a. 1991** Banker, R. ; Kauffmann, R. ; Kumar, R.: An Empirical Test of Object-Based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment. In: *Journal of Management Information Systems* 8 (1991), Nr. 3, S. 127–150

4 **Basili und Weiss 1984** Basili, R. ; Weiss, D. M.: A Methodology for Collecting Valid Software Engineering Data. In: *IEEE Transactions on Software Engineering* 10 (1984), November, Nr. 6, S. 728–738

5 **Basili und Turner 1975** Basili, V. ; Turner, J.: Iterative Enhancement: A Practical Technique for Software Development. In: *IEEE Transactions on Software Engineering* 1 (1975), Dezember, Nr. 4, S. 390–396

6 **Bass u. a. 2003** Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture in Practice*. 2nd ed. Addison Wesley, 2003

7 **Beck 2000** Beck, Kent: *Extreme Programming Explained*. Addison-Wesley, 2000 (The XP Series). – ISBN 201-61641-6

8 **Boehm 1981** Boehm, B.: *Software Engineering Economics*. Prentice Hall, 1981

9 **Boehm und Turner 2003** Boehm, B. ; Turner, R.: Using risk to balance agile and plan-driven methods. In: *IEEE Computer* 36 (2003), Juni, Nr. 6, S. 57–66

0 **Boehm u. a. 1995** Boehm, Barry ; Clark, Bradford ; Horowitz, Ellis ; Madachy, Ray ; Shelby, Richard ; Westland, Chris: Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. In: *Annals of Software Engineering* (1995)

1 **Boehm 1979** Boehm, Barry W.: Guidelines for verifying and validating software requirements and design specification. In: *EURO IFIP 79*, 1979, S. 711–719

2 **Boehm 1988** Boehm, Barry W.: A spiral model of software development and enhancement. In: *IEEE Computer* 21 (1988), Mai, Nr. 5, S. 61–72

3 **Boehm u. a. 2000** Boehm, Barry W. ; Abts, Chris ; Brown, A. W. ; Chulani, Sunita ; Clark, Bradford K. ; Horowitz, Ellis ; Madachy, Ray ; Reifer, Donald ; Steece, Bert: *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000

4 **Buschmann u. a. 1996** Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-oriented Software Architecture: A System of Patterns*. Bd. 1. Wiley, 1996

5 **Chidamber und Kemerer 1994** Chidamber, S.R. ; Kemerer, C.F.: A Metrics Suite for Object Oriented Design. In: *IEEE Transactions on Software Engineering* 20 (1994), Nr. 6, S. 476–493

6 **Clements u. a. 2002** Clements, Paul ; Bachmann, Felix ; Bass, Len ; Garlan, David ; Ivers, James ; Little, Reed ; Nord, Robert ; Stafford, Judith: *Documenting Software Architecture*. Boston : Addison-Wesley, 2002

7 **Clements und Northrop 2001** Clements, Paul ; Northrop, Linda M.: *Software Product Lines : Practices and Patterns*. Addison Wesley, August 2001. – ISBN 0201703327

8 **Cobb und Mills 1990** Cobb, R. H. ; Mills, H. D.: Engineering Software Under Statistical Quality Control. In: *IEEE Software* 7 (1990), Nr. 6, S. 44–54

9 **Endres und Rombach 2003** Endres, Albert ; Rombach, Dieter: *A Handbook of Software and Systems Engineering*. Addison Wesley, 2003

0 **Fenton und Pfleeger 1998** Fenton, N. ; Pfleeger, S.: *Software Metrics: A Rigorous & Practical Approach*. 2nd. London : International Thomson Computer Press, 1998

1 **Gamma u. a. 2003** Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Desig Patterns—Elements of Reusable Object-Oriented Software*. Addison Wesley, 2003

2 **Garlan u. a. 1995** Garlan, D. ; Allen, R. ; Ockerbloom, J.: Architectural Mismatch: Why Reuse is So Hard. In: *IEEE Software* 12 (1995), November, Nr. 6, S. 17–26

3 **Gornik 2001** Gornik, David: IBM Rational Unified Process: Best Practices for Software Development Teams / IBM Rational Software. 2001 (TP026B, Rev 11/01). – White Paper

4 **Halstead 1977** Halstead, Maurice H.: Elements of Software Science. In: *Operating, and Programming Systems Series* 7 (1977)

5 **Hofmeister u. a. 2000** Hofmeister, Christine ; Nord, Robert ; Soni, Dilip: *Applied Software Architecture*. Addison Wesley, 2000 (Object Technology Series)

6 **Humphrey 1995** Humphrey, Watts S.: *A Discipline For Software Engineering*. Addison-Wesley, 1995 (SEI series in software engineering)

7 **IEEE P1471 2002** : *IEEE Recommended Practice for Architectural Description of Software-intensive Systems—Std. 1471-2000*. ISBN 0-7381-2518-0, IEEE, New York, NY, USA. 2002

8 **Jones 1995** Jones, Capers: Backfiring: Converting Lines of Code to Function Points. In: *IEEE Computer* 28 (1995), November, Nr. 11, S. 87–88

9 **Jones 1996** Jones, Capers: Software Estimating Rules of Thumb. In: *IEEE Computer* 29 (1996), March, Nr. 3, S. 116–118

0 **Kauffman und Kumar 1993** Kauffman, R. ; Kumar, R.: Modeling Estimation Expertise in Object Based ICASE Environments / Stern School of Business, New York University. Januar 1993. – Report

1 **Kemerer 1987** Kemerer, Chris F.: An Empirical Validation of Software Cost Estimation Models. In: *Comm. ACM* 30 (1987), May, Nr. 5

2 **Kemerer und Porter 1992** Kemerer, Chris F. ; Porter, Benjamin S.: Improving the Reliability of Function Point Measurement: An Empirical Study. In: *TSE* 18 (1992), Nov., Nr. 11

3 **Kruchten 1998** KRUCHTEN, Phillipe: *The Rational Unified Process: An Introduction*. Reading, Mass.: Addison-Wesley, 1998

4 **Lienert 1973** LIENERT, G.A.: *Verteilungsfreie Methoden in der Biostatistik*. Meisenheim am Glan, Germany : Verlag Anton Hain, 1973. – wird leider nicht mehr aufgelegt

5 **McCabe 1976** MCCABE, T.: A Software Complexity Measure. In: *IEEE Transactions on Software Engineering* 2 (1976), Nr. 4, S. 308–320

6 **Mills u. a. 1987** MILLS, H.D. ; DYER, M. ; LINGER, R.: Cleanroom Software Engineering. In: *IEEE Software* 4 (1987), September, Nr. 5, S. 19–25

7 **Parker 1995** PARKER, Burton G.: *Data Management Maturity Model*. July 1995

8 **Pressman 1997** PRESSMAN, Roger: *Software Engineering – A Practioner's Approach*. Vierte Ausgabe. McGraw-Hill, 1997

9 **Royce 1970** ROYCE, W.: Managing the Development of Large Software Systems. In: *Proceedings Westcon*, IEEEPress, 1970, S. 328–339

0 **Sneed 2004** SNEED, Harry: *Vorlesungsskriptum Software-Engineering*. Uni Regensburg: Wirtschaftsinformatik (Veranst.), 2004

1 **Sommerville 2004** SOMMERVILLE, Ian: *Software Engineering*. Addison-Wesley, 2004

2 **Symons 1988** SYMONS, C. R.: Function Point Analysis: Difficulties and Improvements. In: *TSE* 14 (1988), Jan., Nr. 1, S. 2–11

3 **Szyperski u. a. 2002** SZYPERSKI, Clemens ; GRUNTZ, Dominik ; MURER, Stephan: *Component Software*. Second edition. Addison-Wesley, 2002. – ISBN 0-201-74572-0

4 **Winner u. a. 1991** WINNER, B.J. ; BROWN, Donald R. ; MICHELS, Kenneth M.: *Statistical Principles in Experimental Design*. 3rd edition. McGraw-Hill, 1991 (Series in Psychology)